

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/351405518>

An Introduction to Decentralized Finance (DeFi)

Article in Complex Systems Informatics and Modeling Quarterly · April 2021

DOI: 10.7250/csimg.2021-26.03

CITATIONS

4

READS

5,299

3 authors, including:



Johannes Rude Jensen

Copenhagen Business School

16 PUBLICATIONS 23 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



The normative and philosophical implications of Blockchain based cryptocurrency [View project](#)

An Introduction to Decentralized Finance (DeFi)

Johannes Rude Jensen^{1,2*}, Victor von Wachter¹, and Omri Ross^{1,2}

¹ Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

² eToroX Labs, Copenhagen, Denmark

johannesrudejensen@gmail.com, victor.vonwachter@di.ku.dk, omri@di.ku.dk

Abstract. Decentralized financial applications (DeFi) are a new breed of consumer-facing financial applications composed as smart contracts, deployed on permissionless blockchain technologies. In this article, we situate the DeFi concept in the theoretical context of permissionless blockchain technology and provide a taxonomical overview of agents, incentives and risks. We examine the key market categories and use-cases for DeFi applications today and identify four key risk groups for potential stakeholders contemplating the advantages of decentralized financial applications. We contribute novel insights into a rapidly emerging field, with far-reaching implications for the financial services.

Keywords: Blockchain, Decentralized Finance, DeFi, Smart Contracts.

1 Introduction

Decentralized financial applications, colloquially referred to as ‘DeFi’, are a new type of open financial applications deployed on publicly accessible, permissionless blockchains. A rapid surge in the popularity of these applications saw the total value of the assets locked in DeFi applications (TVL) grow from \$675mn at the outset of 2020 to an excess of \$40bn towards the end of first quarter in the following year[†]. While scholars within the information systems and management disciplines recognize the novelty and prospective impact of blockchain technologies, theoretical or empirical work on DeFi remains scarce [1]. In this short article, we provide a conceptual introduction to ‘DeFi’ situated in the theoretical context of permissionless blockchain technology. We introduce a taxonomy of agents, roles, incentives, and risks in DeFi applications and present four potential sources of complexity and risk.

This article extends the previous publication on managing risk in DeFi[‡] and is structured as follows. Section 2 introduces the permissionless blockchain technology and decentralized finance. Section 3 presents DeFi application taxonomy. An overview of popular DeFi application

* Corresponding author

© 2021 Johannes Rude Jensen, Victor von Wachter, and Omri Ross. This is an open access article licensed under the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0>).

Reference: J. R. Jensen, V. von Wachter, and O. Ross, “An Introduction to Decentralized Finance (DeFi),” Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 26, pp. 46–54, 2021. Available: <https://doi.org/10.7250/csimq.2021-26.03>

Additional information. Author ORCID iD: J. R. Jensen – <https://orcid.org/0000-0002-7835-6424>, V. von Wachter – <https://orcid.org/0000-0003-4275-3660>, and O. Ross – <https://orcid.org/0000-0002-0384-1644>. PII S225599222100150X. Received: 18 February 2021. Accepted: 15 April 2021. Available online: 30 April 2021.

[†] <https://defipulse.com/>

[‡] <http://ceur-ws.org/Vol-2749/short3.pdf>

categories is given in Section 4. The risks in decentralized finance are discussed in Section 5. Section 6 concludes the paper.

2 Permissionless Blockchain Technology and Decentralized Finance

The implications and design principles for blockchain and distributed ledger technologies have generated a growing body of literature in the information systems (IS) genre [2]. Primarily informed by the commercial implications of smart contract technology, scholars have examined the implications for activities in the financial services such as the settlement and clearing of ‘tokenized’ assets [3] the execution and compilation of financial contracts [4]–[6], complexities in supply-chain logistics [7] and beyond. A blockchain is a type of distributed database architecture in which a decentralized network of stakeholders maintains a singleton state machine. Transactions in the database represent state transitions disseminated amongst network participants in ‘blocks’ of data. The correct order of the blocks containing the chronological overview of transactions in the database is maintained with the use of cryptographical primitives, by which all stakeholders can manually verify the succession of blocks.

A network consensus protocol defines the rules for what constitutes a legitimate transaction in the distributed database. In most cases, consensus protocols are rigorous game-theoretical mechanisms in which network participants are economically incentivized to promote network security through rewards and penalties for benevolent or malicious behavior [8]. Scholars typically differentiate between ‘permissioned’ and ‘permissionless’ blockchains. Permissionless blockchains are open environments accessible by all, whereas permissioned blockchains are inaccessible for external parties not recognized by a system administrator [2]. Recent implementations of the technology introduces a virtual machine, the state of which is maintained by the nodes supporting the network. The virtual machine is a simple stack-based architecture, in which network participants can execute metered computations denominated in the native currency format. Because all ‘nodes’ running the blockchain ‘client’ software must replicate the computations required for a program to run, computational expenditures are priced on the open market. This design choice is intended to mitigate excessive use of resources leading to network congestion or abuse.

Network participants pass instructions to the virtual machine in a higher-level programming language, the most recent generations of which is used to write programs, referred to as *smart contracts*. Because operations in the virtual machine are executed in a shared state, smart contracts are both transparent and *stateful*, meaning that any application deployed as a smart contract executes deterministically. This ensures that once a smart contract is deployed, it will execute exactly as instructed.

3 DeFi Agent Taxonomy

We denote the concept: ‘DeFi application’ as an arrangement of consumer-facing smart contracts, executing a predefined business logic within the transparent and deterministic computational environment afforded by a permissionless blockchain technology. Blockchain technology is the core infrastructure layer (see Figure 1) storing transactions securely and providing game-theoretic consensus through the issuance of a native asset. As a basic financial function, standardized smart contracts are utilized to create base assets in the asset layer. These assets are utilized as basis for more complex financial instruments in the application layer. In the application layer, DeFi applications are deployed as sophisticated smart contracts and thus execute a given business logic deterministically. Contemporary DeFi applications provide a range of financial services within trading, lending, derivatives, asset management and insurance services. Aggregators source services from multiple applications, largely to provide the best rates across the ecosystem. Finally, user friendly frontends combine the applications and build a service similar to today’s banking apps. In contrast to traditional banking services, in a

blockchain-based technology stack, users interact directly with the application independent of any intermediary service provider.

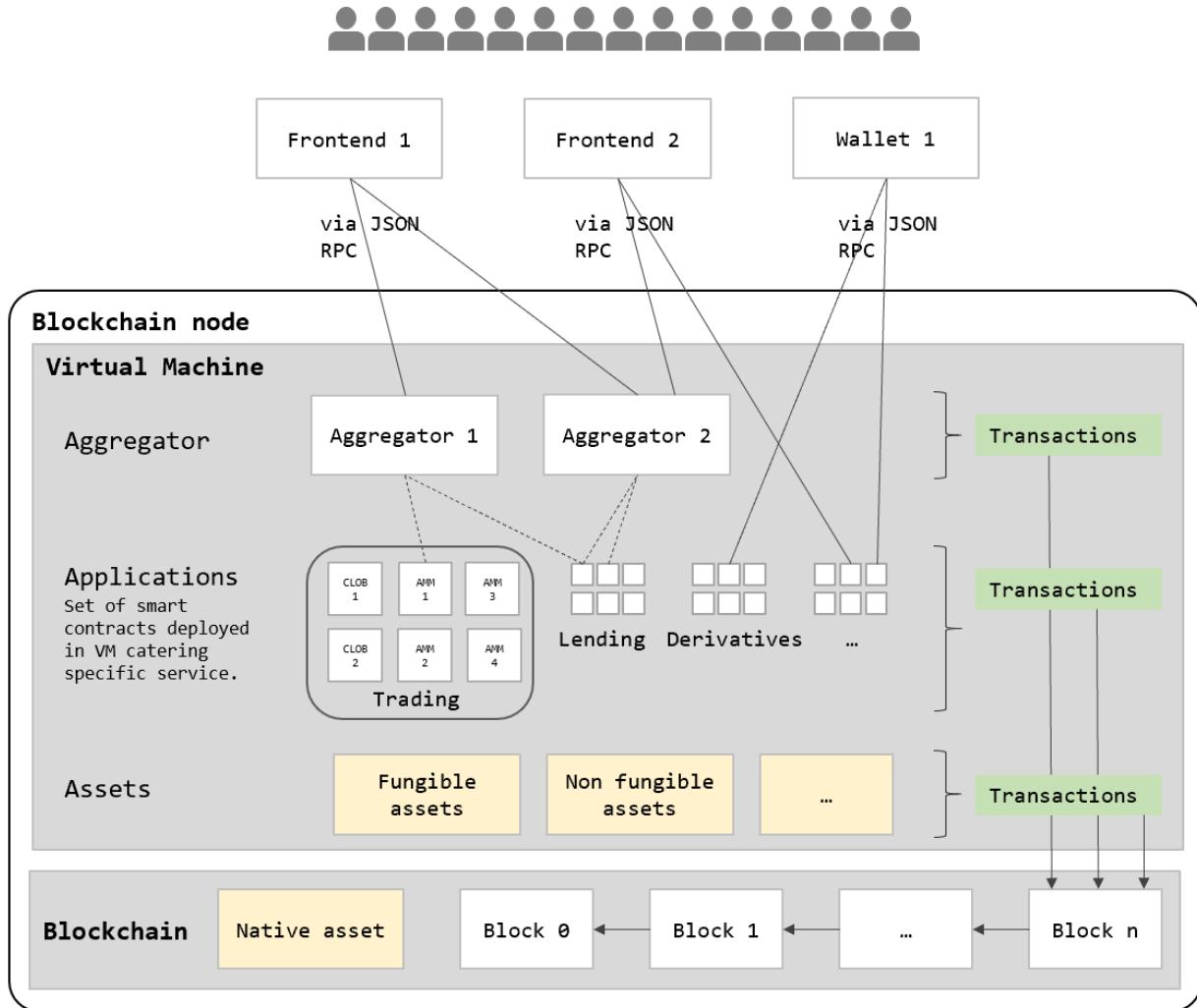


Figure 1. DeFi applications on permissionless blockchain

The metered pricing of computational resources on permissionless blockchains means that DeFi applications are constrained by the computational resources they can use. Application designers seek to mitigate the need for the most expensive operations, such as storing big amounts of data or conducting sophisticated calculations, in the effort of reducing the level of complexity required to execute the service that their application provides.

Because the resources required for interacting with a smart contract are paid by the user, DeFi application designers employ an innovative combination of algorithmic financial engineering and game theory to ensure that all stakeholders of their application are sufficiently compensated and incentivized. In Table 1, we introduce a taxonomy for the different types of agents and their roles in contemporary DeFi applications. We highlight the incentives for participation and key risks associated with each role.

Owing to the original open-source ethos of blockchain technology, application designers are required to be transparent and build ‘open’ and accessible applications, in which users can take ownership and participate in decision-making processes, primarily concerning new features or changes to the applications. As a reaction to these demands, application designers often issue and distribute so-called *governance tokens*. Governance tokens are fungible units held by users, which allocates voting power in majority voting-schemes [9]. Much like traditional equities, governance tokens trade on secondary markets which introduces the opportunity for capital

formation for early stakeholders and designers of successful applications. By distributing governance tokens, application designers seek to disseminate value to community members while retaining enough capital to scale development of the application by selling inventory over multiple years.

Table 1. Agent classification, incentives, and key risks

Agent:	Role:	Incentives for participation:	Key risk:
Users	Utilizing the application	Profits, credit, exposure and governance token	Market risk, technical risk
Liquidity Providers	Supply capital to the application in order to ensure liquidity for traders or borrowers	Protocol fees, governance token	Systemic economic risk, technical risk, regulatory risk, opportunity costs of capital
Arbitrageurs	Return the application to an equilibrium state through strategic purchasing and selling of assets	Arbitrage profits	Market risk, network congestion and transaction fees
Application Designers (Team and Founders)	Design, implement and maintain the application	Governance token appreciation	Software bugs

The generalized agent classification demonstrated in Table 1 is applicable to a wide area of DeFi applications providing peer-to-peer financial services on blockchain technology including, trading, lending, derivatives and asset management. In the following section, we dive into a number of recent use cases, examining the most recently popular categories of applications.

4 An Overview of Popular DeFi Application Categories

The development principles presented above have been implemented in a number of live applications to date. In this section, we provide a brief overview of the main categories of DeFi applications.

4.1 Decentralized Exchanges and Automated Market Makers

Facilitating the decentralized exchange of assets requires an efficient solution for matching counterparties with the desire to sell or purchase a given asset for a certain price, a process known as *price-discovery*. Early implementations of decentralized exchanges on permissionless blockchain technologies successfully demonstrated the feasibility of executing decentralized exchange of assets on permissionless blockchain technology, by imitating the conventional central limit order book (CLOB) design. However, for reasons stipulated below, this proved infeasible and expensive at scale.

First, in the unique cost structure of the blockchain based virtual machine format [10], traders engaging with an application, pay fees corresponding to the complexity of the computation and the amount of storage required for the operation they wish to compute. Because the virtual machine is replicated on all active nodes, storing even small amounts of data is exceedingly expensive. Combined with a complex matching logic required to maintain a liquid orderbook, computing fees rapidly exceeded users' willingness to trade.

Second, as 'miners' pick transactions for inclusion in the next block by the amount of computational fees attached to the transaction, it is possible to front-run state changes to the decentralized orderbook by attaching a large computational fee to a transaction including a trade,

which pre-emptively exploits the next state change of the orderbook, thus profiting through arbitrage on a deterministic future state [11].

Subsequent iterations of decentralized exchanges addressed these issues by storing the state of the orderbook separately, using the blockchain only to compute the final settlement [12]. Nevertheless, problems with settlement frequency persisted, as these implementations introduced complex coordination problems between orderbook storage providers, presenting additional risk vectors to storage security. Motivated by the shortcomings of the established CLOB design a generation of blockchain specific ‘automated’ market makers (AMMs) presents a new approach to blockchain enabled market design.

By pooling available liquidity in trading pairs or groups, AMMs eliminate the need for the presence of buyers and sellers at the same time, facilitating relatively seamless trade execution without compromising the deterministic integrity of the computational environment afforded by the blockchain. Trading liquidity is provided by ‘liquidity providers’ which lock crypto assets in the pursuit of trading fee returns.

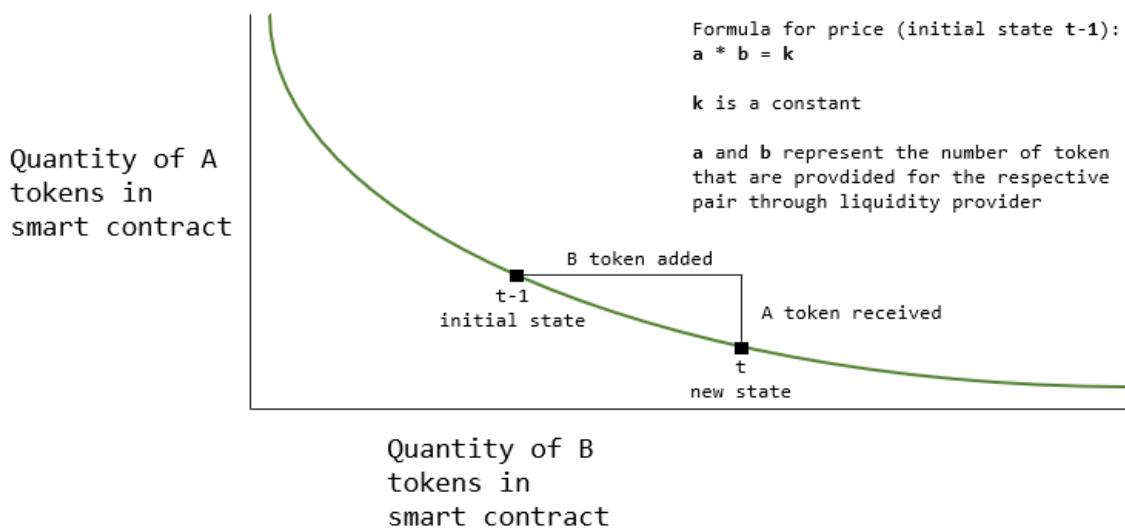


Figure 2. AMM Price Discovery Function

While the primary context for the formal literature on blockchain based AMM has been provided by Angeris and Chitra *et al.* [13]–[15] the field has attracted new work on adjacent topics such as liquidity provisioning [16]–[18] and token weighted voting systems [19].

4.2 Peer-to-Peer Lending and Algorithmic Money Markets

The ‘money markets’ to borrow and lend capital with corresponding interest payments occupy an important role in the traditional financial service. Within DeFi, borrowing and lending applications are amongst the largest segments of financial applications with \$7bn total value locked[§] at the end of 2020. In borrowing/lending protocols agents with excess capital can lend crypto assets (‘liquidity providers’) to a peer-to-peer protocol receiving continuous interest payments. Consequently, a borrower can borrow crypto assets and pays an interest rate. Given the pseudonymous nature of blockchain technology, it is not possible to borrow funds purely on credit. To borrow funds, the borrowing agent has to ‘overcollateralize’ a loan, by providing another crypto assets exceeding the dollar value of the loan to the smart contract. The smart contract then issues a loan relative to 70–90% of the value of the collateral assets. Should the

[§] <https://defipulse.com/>

value of the collateral assets drop below the value of the outstanding loan, the smart contract automatically auctions away the collateral on a decentralized exchange at a profit. The interest rate is algorithmically set by the relative supply and demand for each specific crypto asset. Initially pioneered by the MakerDAO ** application, several protocols are now accessible providing similar services with novel interests rate calculations or optional insurance properties, currently presiding over a \$7bn crypto assets under management.

4.3 Derivatives

Blockchain-based financial contracts (derivatives) are one of the fastest growing market segments in DeFi. Here, application designers seek to make traditional financial derivatives such as *options*, *futures* and other kinds of *synthetic* contracts available to the broader DeFi ecosystem. A futures contract stipulates a sale of an asset at a specified price with an expiry date, an option contract stipulates the *right* but not the obligation to sell or purchase an asset at a specific price.

As in traditional finance both financial services can be used as insurance against market movements as well as speculation on prices. Recently, a new segment of ‘synthetic’ assets has entered the market in the form of tokens pegged to an external price, commonly tracking the price of commodities (e.g., gold) or stocks (e.g., Tesla). A user can create such synthetic asset by collateralized crypto assets in a smart contract similar to how a decentralized lending is computed. The synthetic asset tracks an external price feed (‘oracle’) which is provided to the blockchain. However, external price feeds are prone to technical issues and coordination problems leading to staleness in case of network congestions or fraudulent manipulation [20].

4.4 Automated Asset Management

The traditional practice of ‘asset management’ in the financial services industry consists primarily of the practice of allocating financial assets such as to satisfy the long-term financial objectives of an institution or an individual. As the reader will have noted above, there are an increasing number of DeFi applications, all of which operate algorithmically without human intervention. This means that the DeFi markets operate around the clock and are impossible to manage

The two main use cases for automated asset managers are ‘yield aggregators’ and traditional crypto asset indices. Utilizing the interoperability and automation of blockchain technology, ‘yield aggregators’ are smart contract protocols allocating crypto assets according to predefined rules, often with the goal of maximizing yield whilst controlling risk. Users typically allocate assets to a protocols, which automatically allocates assets across applications in order to optimize the aggregate returns, while rebalancing capital allocations on an ongoing basis.

Indices, on the other hand, offer a broad exposure to crypto assets akin to the practice of ‘passive’ investing. These applications track a portfolio of crypto assets by automatically purchasing these assets and holding them within the smart contract. Equivalent to exchange traded funds (ETFs), stakeholders purchase ownership of the indices by buying a novel token, granting them the algorithmic rights over a fraction of the total assets held within the smart contract^{††}.

5 Identifying and Managing Risk in Decentralized Finance

In this section, we identify and evaluate four risk factors which are likely to introduce new complexities for stakeholders involved with DeFi applications.

** <https://makerdao.com/>

†† blockchain-in-asset-management.pdf (pwc.co.uk)

5.1 Software Integrity and Security

Owing to the deterministic nature of permissionless blockchain technology, applications deployed on as smart contracts are subject to excessive security risks, as any signed transaction remains permanent once included in a block. The irreversible or, ‘immutable’ nature of transactions in a blockchain network has led to significant loss of capital on multiple occasions, most frequently as a result of coding errors, sometimes relating to even the most sophisticated aspects virtual machine and programming language semantics [21]. DeFi applications rely on the integrity of smart contracts and the underlying blockchain. Risk is further enforced through uncertainties in future developments and the novelty of the technology.

5.2 Transaction Costs and Network Congestion

To mitigate abusive or excessive use of the computational resources available on the network, computational resources required to interact with smart contracts are metered. This creates a secondary market for transactions, in which users can outbid each other by attaching transaction fees in the effort of incentivizing miners to select their transaction for inclusion in the next block [11]. In times of network congestion, transactions can remain in a pending state, which ultimately results in market inefficiency and information delays.

Furthermore, in these times, complex transactions can cost up to hundreds of dollars, making potential adjustments to the state costly.^{††} While intermediary service providers occasionally choose to subsidize protocol transaction fees^{§§}, application fees are in near all cases paid by the user interacting with the DeFi application.

Because application designers seek to lower the aggregate transaction costs, protocol fees, slippage or impermanent loss through algorithmic financial modelling and incentive alignment, stakeholders must carefully observe the state of the blockchain network. If a period of network congestion coincides with a period of volatility, the application design may suddenly impose excessive fees or penalties on otherwise standard actions such as withdrawing or adding funds to a lending market [20].

5.3 Participation in Decentralized Governance

Responding to implications of the historically concentrated distribution of native assets amongst a small minority of stakeholders, DeFi application designers increasingly rely on a gradual distribution of fungible governance-tokens in the attempt at adequately ‘decentralizing’ decision-making processes [9].

While the distribution of governance tokens remains fairly concentrated amongst a small group of colluding stakeholders, the gradual distribution of voting-power to liquidity providers and users will result in an increasingly long-tailed distribution of governance tokens. Broad distributions of governance tokens may result in adversarial implications of a given set of governance outcomes, for stakeholders who are not sufficiently involved in monitoring the governance process [19].

5.4 Application Interoperability and Systemic Risks

A key value proposition for DeFi applications is the high level of interoperability between applications. As most applications are deployed on the Ethereum blockchain, users can transact seamlessly between different applications with settlement times rarely exceeding a few minutes. This facilitates rapid capital flows between old and new applications on the network. While interoperability is an attractive feature for any set of financial applications, tightly coupled and

^{††} <https://etherscan.io/gastracker>

^{§§} Coinbase.com

complex liquidity systems can generate an excessive degree of financial integration, resulting in systemic dependencies between applications [22].

This factor is exacerbated by the often complex and heterogeneous methodologies for the computation of exposure, debt, value, and collateral value that DeFi application designers have used to improve their product. An increasing degree of contagion between applications may introduce systemic risks, as a sudden failure or exploit in one application could ripple throughout the network, affecting stakeholders across the entire ecosystem of applications.

The primary example of this dynamic can be demonstrated by the computation of ownership in so-called liquidity pools used by traders utilizing AMM smart contracts. When providing liquidity in the form of crypto assets to a decentralized exchange, liquidity providers receive ‘liquidity shares’ redeemable for a proportional share of the liquidity pool, together with the accumulated fees generated through trading.

As liquidity shares are typically transferable and fungible IOU tokens representing fractional ownership of a liquidity pool, this has led to the emergence of secondary markets for liquidity shares. Providing liquidity in the form of IOU tokens, to these secondary market creates additional (3rd generation) liquidity shares, generating additional fees for the liquidity provider. As a consequence of the increasingly integrated market for liquidity shares, a rapid depreciation of the source asset for the liquidity shares may trigger a sequence of cascading liquidations, as the market struggles to price in any rapid changes in the price of the source asset [20], [22], [23].

6 Conclusion: Is DeFi The Future of Finance?

In this article, we have examined the potential implications, complexities and risks associated with the proliferation of consumer facing DeFi applications. While DeFi applications deployed on permissionless blockchains present a radical potential for transforming consumer facing financial services, the risks associated with engaging with these applications remain salient. Future stakeholder contemplating an engagement with these applications ought to consider and evaluate key risks prior to committing or allocating funds to DeFi applications.

Scholars interested in DeFi applications may approach the theme from numerous angles, extending early research on the market design of DeFi applications [14] or issues related to governance tokens [9], [19] and beyond.

Acknowledgments

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 801199.

References

- [1] J. Kolb, M. Abdelbaky, R. H. Katz, and D. E. Culler, “Core Concepts, Challenges, and future Directions in Blockchain: A centralized Tutorial,” *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–39, 2020. Available: <https://doi.org/10.1145/3366370>
- [2] O. Labazova, “Towards a Framework for Evaluation of Blockchain Implementations,” in *Conference Proceedings of ICIS (2019)*, 2019.
- [3] O. Ross, J. Jensen, and T. Asheim, “Assets under Tokenization: Can Blockchain Technology Improve Post-Trade Processing?” in *Conference Proceedings of ICIS (2019)*, 2019. Available: <https://doi.org/10.2139/ssrn.3488344>
- [4] J. R. Jensen and O. Ross, “Settlement with Distributed Ledger Technology,” in *Conference Proceedings of ICIS (2020)*, 2020.
- [5] B. Egelund-Müller, M. Elsman, F. Henglein, and O. Ross, “Automated Execution of Financial Contracts on Blockchains,” *Bus. Inf. Syst. Eng.*, vol. 59, no. 6, pp. 457–467, 2017. Available: <https://doi.org/10.1007/s12599-017-0507-z>

- [6] O. Ross and J. R. Jensen, “Compact Multiparty Verification of Simple Computations,” in *CEUR Workshop Proceedings*, 2018. Available: <https://doi.org/10.2139/ssrn.3745627>
- [7] B. Düdder and O. Ross, “Timber Tracking: reducing Complexity of Due Diligence by using Blockchain Technology,” *SSRN*, 2017. Available: <https://doi.org/10.2139/ssrn.3015219>
- [8] A. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. Sebastopol, CA: O’Reilly Media, 2018.
- [9] V. von Wachter, J. R. Jensen, and O. Ross, “How Decentralized is the Governance of Blockchain-based Finance? Empirical Evidence from four Governance Token Distributions,” 2020. Available: <https://arxiv.org/abs/2102.10096>
- [10] G. Wood, “Ethereum: A secure decentralized generalized Transaction Ledger EIP 150,” in *Ethereum Project Yellow Paper*, 2014, pp. 1–32.
- [11] P. Daian *et al.*, “Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges,” 2019. Available: <https://arxiv.org/abs/1904.05234>
- [12] W. Warren and A. Bandeali, “0x : An open Protocol for decentralized Exchange on the Ethereum Blockchain.” Available: <https://github.com/0xProject>
- [13] G. Angeris, A. Evans, and T. Chitra, “When does the Tail wag the Dog? Curvature and Market Making,” 2020. Available: <https://arxiv.org/abs/2012.08040>
- [14] G. Angeris, H.-T. Kao, R. Chiang, C. Noyes, and T. Chitra, “An Analysis of Uniswap Markets,” *Cryptoeconomic Systems*, vol. 1, no. 1, 2019. Available: <https://doi.org/10.21428/58320208.c9738e64>
- [15] T. Chitra, “Competitive Equilibria between Staking and on-chain Lending,” *Cryptoeconomic Systems*, vol. 1, no. 1, 2021. Available: <https://doi.org/10.21428/58320208.9ce1cd26>
- [16] J. Aoyagi, “Liquidity Provision by Automated Market Makers,” *SSRN*, 2020. Available: <https://doi.org/10.2139/ssrn.3674178>
- [17] M. Tassy and D. White, “Growth Rate of A Liquidity Provider’s Wealth in XY = c Automated Market Makers,” 2020. Available: https://math.dartmouth.edu/~mtassy/articles/AMM_returns.pdf
- [18] M. Bartoletti, J. H. Chiang, and A. Lluch-Lafuente, “SoK: Lending Pools in Decentralized Finance,” 2020. Available: <https://arxiv.org/abs/2012.13230>
- [19] G. Tsoukalas and B. H. Falk, “Token-Weighted Crowdsourcing,” *Manag. Sci.*, vol. 66, no. 9, pp. 3843–3859, 2020. Available: <https://doi.org/10.1287/mnsc.2019.3515>
- [20] D. Perez, S. M. Werner, J. Xu, and B. Livshits, “Liquidations: DeFi on a Knife-edge,” 2020. Available: <https://arxiv.org/abs/2009.13235>
- [21] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making Smart Contracts Smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS’16)*, pp.254–269, 2016. Available: <https://doi.org/10.1145/2976749.2978309>
- [22] L. Gudgeon, D. Perez, D. Harz, B. Livshits, and A. Gervais, “The Decentralized Financial Crisis,” *Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 1–15, 2020. Available: <https://doi.org/10.1109/CVCBT50464.2020.00005>
- [23] V. von Wachter, J. R. Jensen, and O. Ross, “Measuring Asset Composability as a Proxy for Ecosystem Integration,” in *DeFi Workshop Proceedings of FC’21*, 2021. Available: <https://arxiv.org/abs/2102.04227>

Solidity Documentation

Release 0.8.16

Ethereum

Jun 21, 2022

BASICS

1 Getting Started	3
2 Translations	5
3 Contents	7
3.1 Introduction to Smart Contracts	7
3.2 Installing the Solidity Compiler	15
3.3 Solidity by Example	23
3.4 Layout of a Solidity Source File	45
3.5 Structure of a Contract	48
3.6 Types	51
3.7 Units and Globally Available Variables	88
3.8 Expressions and Control Structures	94
3.9 Contracts	108
3.10 Inline Assembly	148
3.11 Cheatsheet	153
3.12 Using the Compiler	157
3.13 Analysing the Compiler Output	173
3.14 Solidity IR-based Codegen Changes	176
3.15 Layout of State Variables in Storage	181
3.16 Layout in Memory	188
3.17 Layout of Call Data	189
3.18 Cleaning Up Variables	189
3.19 Source Mappings	190
3.20 The Optimizer	191
3.21 Contract Metadata	210
3.22 Contract ABI Specification	214
3.23 Solidity v0.5.0 Breaking Changes	228
3.24 Solidity v0.6.0 Breaking Changes	236
3.25 Solidity v0.7.0 Breaking Changes	239
3.26 Solidity v0.8.0 Breaking Changes	241
3.27 NatSpec Format	244
3.28 Security Considerations	248
3.29 SMTChecker and Formal Verification	255
3.30 Resources	269
3.31 Import Path Resolution	272
3.32 Yul	281
3.33 Style Guide	304
3.34 Common Patterns	325
3.35 List of Known Bugs	332

3.36 Contributing	350
3.37 Solidity Brand Guide	358
3.38 Language Influences	359
Index	361

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.

Solidity is a [curly-bracket language](#) designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python and JavaScript. You can find more details about which languages Solidity has been inspired by in the [language influences](#) section.

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives [security fixes](#). Furthermore, breaking changes as well as new features are introduced regularly. We currently use a 0.y.z version number [to indicate this fast pace of change](#).

Warning: Solidity recently released the 0.8.x version that introduced a lot of breaking changes. Make sure you read [the full list](#).

Ideas for improving Solidity or this documentation are always welcome, read our [contributors guide](#) for more details.

Hint: You can download this documentation as PDF, HTML or Epub by clicking on the versions flyout menu in the bottom-left corner and selecting the preferred download format.

GETTING STARTED

1. Understand the Smart Contract Basics

If you are new to the concept of smart contracts we recommend you to get started by digging into the “Introduction to Smart Contracts” section, which covers:

- *A simple example smart contract* written in Solidity.
- *Blockchain Basics*.
- *The Ethereum Virtual Machine*.

2. Get to Know Solidity

Once you are accustomed to the basics, we recommend you read the “*Solidity by Example*” and “Language Description” sections to understand the core concepts of the language.

3. Install the Solidity Compiler

There are various ways to install the Solidity compiler, simply choose your preferred option and follow the steps outlined on the *installation page*.

Hint: You can try out code examples directly in your browser with the [Remix IDE](#). Remix is a web browser based IDE that allows you to write, deploy and administer Solidity smart contracts, without the need to install Solidity locally.

Warning: As humans write software, it can have bugs. You should follow established software development best-practices when writing your smart contracts. This includes code review, testing, audits, and correctness proofs. Smart contract users are sometimes more confident with code than their authors, and blockchains and smart contracts have their own unique issues to watch out for, so before working on production code, make sure you read the [Security Considerations](#) section.

4. Learn More

If you want to learn more about building decentralized applications on Ethereum, the [Ethereum Developer Resources](#) can help you with further general documentation around Ethereum, and a wide selection of tutorials, tools and development frameworks.

If you have any questions, you can try searching for answers or asking on the [Ethereum StackExchange](#), or our [Gitter channel](#).

**CHAPTER
TWO**

TRANSLATIONS

Community contributors help translate this documentation into several languages. Note that they have varying degrees of completeness and up-to-dateness. The English version stands as a reference.

You can switch between languages by clicking on the flyout menu in the bottom-left corner and selecting the preferred language.

- French
- Indonesian
- Persian
- Japanese
- Korean
- Chinese

Note: We recently set up a new GitHub organization and translation workflow to help streamline the community efforts. Please refer to the [translation guide](#) for information on how to start a new language or contribute to the community translations.

CONTENTS

[Keyword Index](#), [Search Page](#)

3.1 Introduction to Smart Contracts

3.1.1 A Simple Smart Contract

Let us begin with a basic example that sets the value of a variable and exposes it for other contracts to access. It is fine if you do not understand everything right now, we will go into more details later.

Storage Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

The first line tells you that the source code is licensed under the GPL version 3.0. Machine-readable license specifiers are important in a setting where publishing the source code is the default.

The next line specifies that the source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.9.0. This is to ensure that the contract is not compilable with a new (breaking) compiler version, where it could behave differently. *Pragmas* are common instructions for compilers about how to treat the source code (e.g. `pragma once`).

A contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line `uint storedData;` declares a state variable called `storedData` of type `uint` (*unsigned integer* of 256 bits). You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. In this example, the contract defines the functions `set` and `get` that can be used to modify or retrieve the value of the variable.

To access a member (like a state variable) of the current contract, you do not typically add the `this.` prefix, you just access it directly via its name. Unlike in some other languages, omitting it is not just a matter of style, it results in a completely different way to access the member, but more on this later.

This contract does not do much yet apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Anyone could call `set` again with a different value and overwrite your number, but the number is still stored in the history of the blockchain. Later, you will see how you can impose access restrictions so that only you can alter the number.

Warning: Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such are encoded as a different byte array.

Note: All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

Subcurrency Example

The following contract implements the simplest form of a cryptocurrency. The contract allows only its creator to create new coins (different issuance schemes are possible). Anyone can send coins to each other without a need for registering with a username and password, all you need is an Ethereum keypair.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
```

(continues on next page)

(continued from previous page)

```
// to the caller of the function.
error InsufficientBalance(uint requested, uint available);

// Sends an amount of existing coins
// from any caller to an address
function send(address receiver, uint amount) public {
    if (amount > balances[msg.sender])
        revert InsufficientBalance({
            requested: amount,
            available: balances[msg.sender]
        });

    balances[msg.sender] -= amount;
    balances[receiver] += amount;
    emit Sent(msg.sender, receiver, amount);
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type `address`. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to *external accounts*.

The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is equivalent to the following (ignore `external` and `view` for now):

```
function minter() external view returns (address) { return minter; }
```

You could add a function like the above yourself, but you would have a function and state variable with the same name. You do not need to do this, the compiler figures it out for you.

The next line, `mapping (address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The `mapping` type maps addresses to *unsigned integers*.

Mappings can be seen as `hash tables` which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. Record what you added to the mapping, or use it in a context where this is not needed. Or even better, keep a list, or use a more suitable data type.

The `getter function` created by the `public` keyword is more complex in the case of a mapping. It looks like the following:

```
function balances(address account) external view returns (uint) {
    return balances[account];
}
```

You can use this function to query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares an “*event*”, which is emitted in the last line of the function `send`. Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost. As soon as it is emitted, the listener receives the arguments `from`, `to` and `amount`, which makes it possible to track transactions.

To listen for this event, you could use the following JavaScript code, which uses `web3.js` to create the `Coin` contract object, and any user interface calls the automatically generated `balances` function from above:

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

The *constructor* is a special function that is executed during the creation of the contract and cannot be called afterwards. In this case, it permanently stores the address of the person creating the contract. The `msg` variable (together with `tx` and `block`) is a *special global variable* that contains properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

The functions that make up the contract, and that users and contracts can call are `mint` and `send`.

The `mint` function sends an amount of newly created coins to another address. The `require` function call defines conditions that reverts all changes if not met. In this example, `require(msg.sender == minter);` ensures that only the creator of the contract can call `mint`. In general, the creator can mint as many tokens as they like, but at some point, this will lead to a phenomenon called “overflow”. Note that because of the default *Checked arithmetic*, the transaction would revert if the expression `balances[receiver] += amount;` overflows, i.e., when `balances[receiver] + amount` in arbitrary precision arithmetic is larger than the maximum value of `uint` ($2^{**256} - 1$). This is also true for the statement `balances[receiver] += amount;` in the function `send`.

Errors allow you to provide more information to the caller about why a condition or operation failed. Errors are used together with the *revert statement*. The `revert` statement unconditionally aborts and reverts all changes similar to the `require` function, but it also allows you to provide the name of an error and additional data which will be supplied to the caller (and eventually to the front-end application or block explorer) so that a failure can more easily be debugged or reacted upon.

The `send` function can be used by anyone (who already has some of these coins) to send coins to anyone else. If the sender does not have enough coins to send, the `if` condition evaluates to true. As a result, the `revert` will cause the operation to fail while providing the sender with error details using the `InsufficientBalance` error.

Note: If you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the record that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By using events, you can create a “blockchain explorer” that tracks transactions and balances of your new coin, but you have to inspect the coin contract address and not the addresses of the coin owners.

3.1.2 Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, [hashing](#), [elliptic-curve cryptography](#), [peer-to-peer networks](#), etc.) are just there to provide a certain set of features and promises for the platform. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon’s AWS works internally in order to use it?

Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is being applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

Blocks

One major obstacle to overcome is what (in Bitcoin terms) is called a “double-spend attack”: What happens if two transactions exist in the network that both want to empty an account? Only one of the transactions can be valid, typically the one that is accepted first. The problem is that “first” is not an objective term in a peer-to-peer network.

The abstract answer to this is that you do not have to care. A globally accepted order of the transactions will be selected for you, solving the conflict. The transactions will be bundled into what is called a “block” and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time and that is where the word “blockchain” derives from. Blocks are added to the chain in rather regular intervals - for Ethereum this is roughly every 17 seconds.

As part of the “order selection mechanism” (which is called “mining”) it may happen that blocks are reverted from time to time, but only at the “tip” of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

Note: Transactions are not guaranteed to be included in the next block or any specific future block, since it is not up to the submitter of a transaction, but up to the miners to determine in which block the transaction is included.

If you want to schedule future calls of your contract, you can use a smart contract automation tool or an oracle service.

3.1.3 The Ethereum Virtual Machine

Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called “nonce”).

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in “Wei” to be exact, 1 ether is 10^{**18} wei) which can be modified by sending transactions that include Ether.

Transactions

A transaction is a message that is sent from one account to another account (which might be the same or empty, see below). It can include binary data (which is called “payload”) and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the transaction does not have a recipient or the recipient is set to null), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the “nonce”). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

Note: While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

Gas

Upon creation, each transaction is charged with a certain amount of **gas** that has to be paid for by the originator of the transaction (`tx.origin`). While the EVM executes the transaction, the gas is gradually depleted according to specific rules. If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which ends execution and reverts all modifications made to the state in the current call frame.

This mechanism incentivizes economical use of EVM execution time and also compensates EVM executors (i.e. miners / stakers) for their work. Since each block has a maximum amount of gas, it also limits the amount of work needed to validate a block.

The **gas price** is a value set by the originator of the transaction, who has to pay `gas_price * gas` up front to the EVM executor. If some gas is left after execution, it is refunded to the transaction originator. In case of an exception that reverts changes, already used up gas is not refunded.

Since EVM executors can choose to include a transaction or not, transaction senders cannot abuse the system by setting a low gas price.

Storage, Memory and the Stack

The Ethereum Virtual Machine has three areas where it can store data: storage, memory and the stack.

Each account has a data area called **storage**, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage. Because of this cost, you should minimize what you store in persistent storage to what the contract needs to run. Store data like derived calculations, caching, and aggregates outside of the contract. A contract can neither read nor write to any storage apart from its own.

The second data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words or on slices of memory (or other byte arrays). The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

For a complete list, please see the [list of opcodes](#) as part of the inline assembly documentation.

Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions “bubble up” the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has finished execution, it can return data which will be stored at a location in the caller’s memory preallocated by the caller. All such calls are fully synchronous.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls. Furthermore, only 63/64th of the gas can be forwarded in a message call, which causes a depth limit of a little less than 1000 in practice.

Delegatecall / Callcode and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context (i.e. at the address) of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the “library” feature in Solidity: Reusable library code that can be applied to a contract’s storage, e.g. in order to implement a complex data structure.

Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement *events*. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in *bloom filters*, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain (so-called “light clients”) can still find these logs.

Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address as a transaction would). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

Deactivate and Self-destruct

The only way to remove code from the blockchain is when a contract at that address performs the **selfdestruct** operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

Warning: Even if a contract is removed by `selfdestruct`, it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using `selfdestruct` is not the same as deleting data from a hard disk.

Note: Even if a contract’s code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

Precompiled Contracts

There is a small set of contract addresses that are special: The address range between 1 and (including) 8 contains “precompiled contracts” that can be called as any other contract but their behaviour (and their gas consumption) is not defined by EVM code stored at that address (they do not contain code) but instead is implemented in the EVM execution environment itself.

Different EVM-compatible chains might use a different set of precompiled contracts. It might also be possible that new precompiled contracts are added to the Ethereum main chain in the future, but you can reasonably expect them to always be in the range between 1 and `0xfffff` (inclusive).

3.2 Installing the Solidity Compiler

3.2.1 Versioning

Solidity versions follow [Semantic Versioning](#). In addition, patch level releases with major release 0 (i.e. 0.x.y) will not contain breaking changes. That means code that compiles with version 0.x.y can be expected to compile with 0.x.z where z > y.

In addition to releases, we provide **nightly development builds** with the intention of making it easy for developers to try out upcoming features and provide early feedback. Note, however, that while the nightly builds are usually very stable, they contain bleeding-edge code from the development branch and are not guaranteed to be always working. Despite our best efforts, they might contain undocumented and/or broken changes that will not become a part of an actual release. They are not meant for production use.

When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes, as well as new features and bug fixes are introduced regularly. We currently use a 0.x version number [to indicate this fast pace of change](#).

3.2.2 Remix

We recommend Remix for small contracts and for quickly learning Solidity.

Access [Remix online](#), you do not need to install anything. If you want to use it without connection to the Internet, go to <https://github.com/ethereum/remix-live/tree/gh-pages> and download the .zip file as explained on that page. Remix is also a convenient option for testing nightly builds without installing multiple Solidity versions.

Further options on this page detail installing commandline Solidity compiler software on your computer. Choose a commandline compiler if you are working on a larger contract or if you require more compilation options.

3.2.3 npm / Node.js

Use `npm` for a convenient and portable way to install `solcjs`, a Solidity compiler. The `solcjs` program has fewer features than the ways to access the compiler described further down this page. The [Using the Commandline Compiler](#) documentation assumes you are using the full-featured compiler, `solc`. The usage of `solcjs` is documented inside its own [repository](#).

Note: The `solc-js` project is derived from the C++ `solc` by using Emscripten which means that both use the same compiler source code. `solc-js` can be used in JavaScript projects directly (such as Remix). Please refer to the `solc-js` repository for instructions.

```
npm install -g solc
```

Note: The commandline executable is named `solcjs`.

The commandline options of `solcjs` are not compatible with `solc` and tools (such as `geth`) expecting the behaviour of `solc` will not work with `solcjs`.

3.2.4 Docker

Docker images of Solidity builds are available using the `solc` image from the `ethereum` organisation. Use the `stable` tag for the latest released version, and `nightly` for potentially unstable changes in the develop branch.

The Docker image runs the compiler executable, so you can pass all compiler arguments to it. For example, the command below pulls the stable version of the `solc` image (if you do not have it already), and runs it in a new container, passing the `--help` argument.

```
docker run ethereum/solc:stable --help
```

You can also specify release build versions in the tag, for example, for the 0.5.4 release.

```
docker run ethereum/solc:0.5.4 --help
```

To use the Docker image to compile Solidity files on the host machine mount a local folder for input and output, and specify the contract to compile. For example.

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --bin /  
→sources/Contract.sol
```

You can also use the standard JSON interface (which is recommended when using the compiler with tooling). When using this interface it is not necessary to mount any directories as long as the JSON input is self-contained (i.e. it does not refer to any external files that would have to be *loaded by the import callback*).

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

3.2.5 Linux Packages

Binary packages of Solidity are available at [solidity/releases](#).

We also have PPAs for Ubuntu, you can get the latest stable version using the following commands:

```
sudo add-apt-repository ppa:ethereum/ethereum  
sudo apt-get update  
sudo apt-get install solc
```

The nightly version can be installed using these commands:

```
sudo add-apt-repository ppa:ethereum/ethereum  
sudo add-apt-repository ppa:ethereum/ethereum-dev  
sudo apt-get update  
sudo apt-get install solc
```

Furthermore, some Linux distributions provide their own packages. These packages are not directly maintained by us, but usually kept up-to-date by the respective package maintainers.

For example, Arch Linux has packages for the latest development version:

```
pacman -S solidity
```

There is also a [snap package](#), however, it is **currently unmaintained**. It is installable in all the [supported Linux distros](#). To install the latest stable version of solc:

```
sudo snap install solc
```

If you want to help testing the latest development version of Solidity with the most recent changes, please use the following:

```
sudo snap install solc --edge
```

Note: The `solc` snap uses strict confinement. This is the most secure mode for snap packages but it comes with limitations, like accessing only the files in your `/home` and `/media` directories. For more information, go to [Demystifying Snap Confinement](#).

3.2.6 macOS Packages

We distribute the Solidity compiler through Homebrew as a build-from-source version. Pre-built bottles are currently not supported.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

To install the most recent 0.4.x / 0.5.x version of Solidity you can also use `brew install solidity@4` and `brew install solidity@5`, respectively.

If you need a specific version of Solidity you can install a Homebrew formula directly from Github.

[View `solidity.rb` commits on Github](#).

Copy the commit hash of the version you want and check it out on your machine.

```
git clone https://github.com/ethereum/homebrew-ethereum.git
cd homebrew-ethereum
git checkout <your-hash-goes-here>
```

Install it using `brew`:

```
brew unlink solidity
# eg. Install 0.4.8
brew install solidity.rb
```

3.2.7 Static Binaries

We maintain a repository containing static builds of past and current compiler versions for all supported platforms at [solc-bin](#). This is also the location where you can find the nightly builds.

The repository is not only a quick and easy way for end users to get binaries ready to be used out-of-the-box but it is also meant to be friendly to third-party tools:

- The content is mirrored to <https://binaries.soliditylang.org> where it can be easily downloaded over HTTPS without any authentication, rate limiting or the need to use git.
- Content is served with correct *Content-Type* headers and lenient CORS configuration so that it can be directly loaded by tools running in the browser.
- Binaries do not require installation or unpacking (with the exception of older Windows builds bundled with necessary DLLs).
- We strive for a high level of backwards-compatibility. Files, once added, are not removed or moved without providing a symlink/redirect at the old location. They are also never modified in place and should always match the original checksum. The only exception would be broken or unusable files with a potential to cause more harm than good if left as is.
- Files are served over both HTTP and HTTPS. As long as you obtain the file list in a secure way (via git, HTTPS, IPFS or just have it cached locally) and verify hashes of the binaries after downloading them, you do not have to use HTTPS for the binaries themselves.

The same binaries are in most cases available on the [Solidity release page on Github](#). The difference is that we do not generally update old releases on the Github release page. This means that we do not rename them if the naming convention changes and we do not add builds for platforms that were not supported at the time of release. This only happens in [solc-bin](#).

The [solc-bin](#) repository contains several top-level directories, each representing a single platform. Each one contains a `list.json` file listing the available binaries. For example in `emscripten-wasm32/list.json` you will find the following information about version 0.7.4:

```
{  
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",  
  "version": "0.7.4",  
  "build": "commit.3f05b770",  
  "longVersion": "0.7.4+commit.3f05b770",  
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",  
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",  
  "urls": [  
    "bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1",  
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDih8zxyqBHGFDRSzVE5CS"  
  ]  
}
```

This means that:

- You can find the binary in the same directory under the name `solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js`. Note that the file might be a symlink, and you will need to resolve it yourself if you are not using git to download it or your file system does not support symlinks.
- The binary is also mirrored at <https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js>. In this case git is not necessary and symlinks are resolved transparently, either by serving a copy of the file or returning a HTTP redirect.
- The file is also available on IPFS at [QmTLs5MuLEWXQkths41HiACoXDih8zxyqBHGFDRSzVE5CS](https://ipfs.io/ipfs/QmTLs5MuLEWXQkths41HiACoXDih8zxyqBHGFDRSzVE5CS).

- The file might in future be available on Swarm at [16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1](https://swarm.cryptonote.com/16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1).
- You can verify the integrity of the binary by comparing its keccak256 hash to `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`. The hash can be computed on the command line using `keccak256sum` utility provided by `sha3sum` or `keccak256()` function from `ethereumjs-util` in JavaScript.
- You can also verify the integrity of the binary by comparing its sha256 hash to `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`.

Warning: Due to the strong backwards compatibility requirement the repository contains some legacy elements but you should avoid using them when writing new tools:

- Use `emscripten-wasm32/` (with a fallback to `emscripten-asmjs/`) instead of `bin/` if you want the best performance. Until version 0.6.1 we only provided `asm.js` binaries. Starting with 0.6.2 we switched to [WebAssembly builds](#) with much better performance. We have rebuilt the older versions for `wasm` but the original `asm.js` files remain in `bin/`. The new ones had to be placed in a separate directory to avoid name clashes.
- Use `emscripten-asmjs/` and `emscripten-wasm32/` instead of `bin/` and `wasm/` directories if you want to be sure whether you are downloading a `wasm` or an `asm.js` binary.
- Use `list.json` instead of `list.js` and `list.txt`. The JSON list format contains all the information from the old ones and more.
- Use <https://binaries.soliditylang.org> instead of <https://solc-bin.ethereum.org>. To keep things simple we moved almost everything related to the compiler under the new `soliditylang.org` domain and this applies to `solc-bin` too. While the new domain is recommended, the old one is still fully supported and guaranteed to point at the same location.

Warning: The binaries are also available at <https://ethereum.github.io/solc-bin/> but this page stopped being updated just after the release of version 0.7.2, will not receive any new releases or nightly builds for any platform and does not serve the new directory structure, including non-`emscripten` builds.

If you are using it, please switch to <https://binaries.soliditylang.org>, which is a drop-in replacement. This allows us to make changes to the underlying hosting in a transparent way and minimize disruption. Unlike the `ethereum.github.io` domain, which we do not have any control over, `binaries.soliditylang.org` is guaranteed to work and maintain the same URL structure in the long-term.

3.2.8 Building from Source

Prerequisites - All Operating Systems

The following are dependencies for all builds of Solidity:

Software	Notes
<code>CMake</code> (version 3.13+)	Cross-platform build file generator.
<code>Boost</code> (version 1.77+ on Windows, 1.65+ otherwise)	C++ libraries.
<code>Git</code>	Command-line tool for retrieving source code.
<code>z3</code> (version 4.8+, Optional)	For use with SMT checker.
<code>cvc4</code> (Optional)	For use with SMT checker.

Note: Solidity versions prior to 0.5.10 can fail to correctly link against Boost versions 1.70+. A possible workaround is

to temporarily rename <Boost install path>/lib/cmake/Boost-1.70.0 prior to running the cmake command to configure solidity.

Starting from 0.5.10 linking against Boost 1.70+ should work without manual intervention.

Note: The default build configuration requires a specific Z3 version (the latest one at the time the code was last updated). Changes introduced between Z3 releases often result in slightly different (but still valid) results being returned. Our SMT tests do not account for these differences and will likely fail with a different version than the one they were written for. This does not mean that a build using a different version is faulty. If you pass `-DSTRICT_Z3_VERSION=OFF` option to CMake, you can build with any version that satisfies the requirement given in the table above. If you do this, however, please remember to pass the `--no-smt` option to `scripts/tests.sh` to skip the SMT tests.

Minimum Compiler Versions

The following C++ compilers and their minimum versions can build the Solidity codebase:

- [GCC](#), version 8+
- [Clang](#), version 7+
- [MSVC](#), version 2019+

Prerequisites - macOS

For macOS builds, ensure that you have the latest version of [Xcode](#) installed. This contains the [Clang C++ compiler](#), the [Xcode IDE](#) and other Apple development tools that are required for building C++ applications on OS X. If you are installing Xcode for the first time, or have just installed a new version then you will need to agree to the license before you can do command-line builds:

```
sudo xcodebuild -license accept
```

Our OS X build script uses the [Homebrew](#) package manager for installing external dependencies. Here's how to [uninstall Homebrew](#), if you ever want to start again from scratch.

Prerequisites - Windows

You need to install the following dependencies for Windows builds of Solidity:

Software	Notes
Visual Studio 2019 Build Tools	C++ compiler
Visual Studio 2019 (Optional)	C++ compiler and dev environment.
Boost (version 1.77+)	C++ libraries.

If you already have one IDE and only need the compiler and libraries, you could install Visual Studio 2019 Build Tools.

Visual Studio 2019 provides both IDE and necessary compiler and libraries. So if you have not got an IDE and prefer to develop Solidity, Visual Studio 2019 may be a choice for you to get everything setup easily.

Here is the list of components that should be installed in Visual Studio 2019 Build Tools or Visual Studio 2019:

- Visual Studio C++ core features
- VC++ 2019 v141 toolset (x86,x64)

- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

We have a helper script which you can use to install all required external dependencies:

```
scripts\install_deps.ps1
```

This will install boost and cmake to the deps subdirectory.

Clone the Repository

To clone the source code, execute the following command:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

If you want to help developing Solidity, you should fork Solidity and add your personal fork as a second remote:

```
git remote add personal git@github.com:[username]/solidity.git
```

Note: This method will result in a prerelease build leading to e.g. a flag being set in each bytecode produced by such a compiler. If you want to re-build a released Solidity compiler, then please use the source tarball on the github release page:

https://github.com/ethereum/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz

(not the “Source code” provided by github).

Command-Line Build

Be sure to install External Dependencies (see above) before build.

Solidity project uses CMake to configure the build. You might want to install `ccache` to speed up repeated builds. CMake will pick it up automatically. Building Solidity is quite similar on Linux, macOS and other Unices:

```
mkdir build
cd build
cmake .. && make
```

or even easier on Linux and macOS, you can run:

```
#note: this will install binaries solc and soltest at /usr/local/bin
./scripts/build.sh
```

Warning: BSD builds should work, but are untested by the Solidity team.

And for Windows:

```
mkdir build  
cd build  
cmake -G "Visual Studio 16 2019" ..
```

In case you want to use the version of boost installed by `scripts\install_deps.ps1`, you will additionally need to pass `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` and `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` as arguments to the call to `cmake`.

This should result in the creation of **solidity.sln** in that build directory. Double-clicking on that file should result in Visual Studio firing up. We suggest building **Release** configuration, but all others work.

Alternatively, you can build for Windows on the command-line, like so:

```
cmake --build . --config Release
```

3.2.9 CMake Options

If you are interested what CMake options are available run `cmake .. -LH`.

SMT Solvers

Solidity can be built against SMT solvers and will do so by default if they are found in the system. Each solver can be disabled by a `cmake` option.

Note: In some cases, this can also be a potential workaround for build failures.

Inside the build folder you can disable them, since they are enabled by default:

```
# disables only Z3 SMT Solver.  
cmake .. -DUSE_Z3=OFF  
  
# disables only CVC4 SMT Solver.  
cmake .. -DUSE_CVC4=OFF  
  
# disables both Z3 and CVC4  
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.2.10 The Version String in Detail

The Solidity version string contains four parts:

- the version number
- pre-release tag, usually set to `develop.YYYY.MM.DD` or `nightly.YYYY.MM.DD`
- commit in the format of `commit.GITHASH`
- platform, which has an arbitrary number of items, containing details about the platform and compiler

If there are local modifications, the commit will be postfixed with `.mod`.

These parts are combined as required by SemVer, where the Solidity pre-release tag equals to the SemVer pre-release and the Solidity commit and platform combined make up the SemVer build metadata.

A release example: `0.4.8+commit.60cc1668.Emscripten clang`.

A pre-release example: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten clang`

3.2.11 Important Information About Versioning

After a release is made, the patch version level is bumped, because we assume that only patch level changes follow. When changes are merged, the version should be bumped according to SemVer and the severity of the change. Finally, a release is always made with the version of the current nightly build, but without the `prerelease` specifier.

Example:

1. The 0.4.0 release is made.
2. The nightly build has a version of 0.4.1 from now on.
3. Non-breaking changes are introduced → no change in version.
4. A breaking change is introduced → version is bumped to 0.5.0.
5. The 0.5.0 release is made.

This behaviour works well with the `version` *pragma*.

3.3 Solidity by Example

3.3.1 Voting

The following contract is quite complex, but showcases a lot of Solidity's features. It implements a voting contract. Of course, the main problems of electronic voting is how to assign voting rights to the correct persons and how to prevent manipulation. We will not solve all problems here, but at least we will show how delegated voting can be done so that vote counting is **automatic and completely transparent** at the same time.

The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually.

The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust.

At the end of the voting time, `winningProposal()` will return the proposal with the largest number of votes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted; // if true, that person already voted
        address delegate; // person delegated to
        uint vote; // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name; // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }
}
```

(continues on next page)

(continued from previous page)

```

address public chairperson;

// This declares a state variable that
// stores a `Voter` struct for each possible address.
mapping(address => Voter) public voters;

// A dynamically-sized array of `Proposal` structs.
Proposal[] public proposals;

// Create a new ballot to choose one of `proposalNames`.
constructor(bytes32[] memory proposalNames) {
    chairperson = msg.sender;
    voters[chairperson].weight = 1;

    // For each of the provided proposal names,
    // create a new proposal object and add it
    // to the end of the array.
    for (uint i = 0; i < proposalNames.length; i++) {
        // `Proposal({...})` creates a temporary
        // Proposal object and `proposals.push(...)`
        // appends it to the end of `proposals`.
        proposals.push(Proposal({
            name: proposalNames[i],
            voteCount: 0
        }));
    }
}

// Give `voter` the right to vote on this ballot.
// May only be called by `chairperson`.
function giveRightToVote(address voter) external {
    // If the first argument of `require` evaluates
    // to `false`, execution terminates and all
    // changes to the state and to Ether balances
    // are reverted.
    // This used to consume all gas in old EVM versions, but
    // not anymore.
    // It is often a good idea to use `require` to check if
    // functions are called correctly.
    // As a second argument, you can also provide an
    // explanation about what went wrong.
    require(
        msg.sender == chairperson,
        "Only chairperson can give right to vote."
    );
    require(
        !voters[voter].voted,
        "The voter already voted."
    );
    require(voters[voter].weight == 0);
    voters[voter].weight = 1;
}

```

(continues on next page)

(continued from previous page)

```

}

/// Delegate your vote to the voter `to`.
function delegate(address to) external {
    // assigns reference
    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "You have no right to vote");
    require(!sender.voted, "You already voted.");

    require(to != msg.sender, "Self-delegation is disallowed.");

    // Forward the delegation as long as
    // `to` also delegated.
    // In general, such loops are very dangerous,
    // because if they run too long, they might
    // need more gas than is available in a block.
    // In this case, the delegation will not be executed,
    // but in other situations, such loops might
    // cause a contract to get "stuck" completely.
    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;

        // We found a loop in the delegation, not allowed.
        require(to != msg.sender, "Found loop in delegation.");
    }

    Voter storage delegate_ = voters[to];

    // Voters cannot delegate to accounts that cannot vote.
    require(delegate_.weight >= 1);

    // Since `sender` is a reference, this
    // modifies `voters[msg.sender]`.
    sender.voted = true;
    sender.delegate = to;

    if (delegate_.voted) {
        // If the delegate already voted,
        // directly add to the number of votes
        proposals[delegate_.vote].voteCount += sender.weight;
    } else {
        // If the delegate did not vote yet,
        // add to her weight.
        delegate_.weight += sender.weight;
    }
}

/// Give your vote (including votes delegated to you)
/// to proposal `proposals[proposal].name`.
function vote(uint proposal) external {
    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "Has no right to vote");
}

```

(continues on next page)

(continued from previous page)

```

require(!sender.voted, "Already voted.");
sender.voted = true;
sender.vote = proposal;

// If `proposal` is out of the range of the array,
// this will throw automatically and revert all
// changes.
proposals[proposal].voteCount += sender.weight;
}

/// @dev Computes the winning proposal taking all
/// previous votes into account.
function winningProposal() public view
    returns (uint winningProposal_)
{
    uint winningVoteCount = 0;
    for (uint p = 0; p < proposals.length; p++) {
        if (proposals[p].voteCount > winningVoteCount) {
            winningVoteCount = proposals[p].voteCount;
            winningProposal_ = p;
        }
    }
}

// Calls winningProposal() function to get the index
// of the winner contained in the proposals array and then
// returns the name of the winner
function winnerName() external view
    returns (bytes32 winnerName_)
{
    winnerName_ = proposals[winningProposal()].name;
}
}

```

Possible Improvements

Currently, many transactions are needed to assign the rights to vote to all participants. Can you think of a better way?

3.3.2 Blind Auction

In this section, we will show how easy it is to create a completely blind auction contract on Ethereum. We will start with an open auction where everyone can see the bids that are made and then extend this contract into a blind auction where it is not possible to see the actual bid until the bidding period ends.

Simple Open Auction

The general idea of the following simple auction contract is that everyone can send their bids during a bidding period. The bids already include sending money / Ether in order to bind the bidders to their bid. If the highest bid is raised, the previous highest bidder gets their money back. After the end of the bidding period, the contract has to be called manually for the beneficiary to receive their money - contracts cannot activate themselves.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Errors that describe failures.

    // The triple-slash comments are so-called natspec
    // comments. They will be shown when the user
    // is asked to confirm a transaction or
    // when an error is displayed.

    /// The auction has already ended.
    error AuctionAlreadyEnded();
    /// There is already a higher or equal bid.
    error BidNotHighEnough(uint highestBid);
    /// The auction has not ended yet.
    error AuctionNotYetEnded();
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();
}
```

(continues on next page)

(continued from previous page)

```

/// Create a simple auction with `biddingTime`  

/// seconds bidding time on behalf of the  

/// beneficiary address `beneficiaryAddress`.  

constructor(  

    uint biddingTime,  

    address payable beneficiaryAddress  

) {  

    beneficiary = beneficiaryAddress;  

    auctionEndTime = block.timestamp + biddingTime;  

}  

/// Bid on the auction with the value sent  

/// together with this transaction.  

/// The value will only be refunded if the  

/// auction is not won.  

function bid() external payable {  

    // No arguments are necessary, all  

    // information is already part of  

    // the transaction. The keyword payable  

    // is required for the function to  

    // be able to receive Ether.  

    // Revert the call if the bidding  

    // period is over.  

    if (block.timestamp > auctionEndTime)  

        revert AuctionAlreadyEnded();  

    // If the bid is not higher, send the  

    // money back (the revert statement  

    // will revert all changes in this  

    // function execution including  

    // it having received the money).  

    if (msg.value <= highestBid)  

        revert BidNotHighEnough(highestBid);  

    if (highestBid != 0) {  

        // Sending back the money by simply using  

        // highestBidder.send(highestBid) is a security risk  

        // because it could execute an untrusted contract.  

        // It is always safer to let the recipients  

        // withdraw their money themselves.  

        pendingReturns[highestBidder] += highestBid;  

    }  

    highestBidder = msg.sender;  

    highestBid = msg.value;  

    emit HighestBidIncreased(msg.sender, msg.value);  

}  

/// Withdraw a bid that was overbid.  

function withdraw() external returns (bool) {  

    uint amount = pendingReturns[msg.sender];  

    if (amount > 0) {

```

(continues on next page)

(continued from previous page)

```

// It is important to set this to zero because the recipient
// can call this function again as part of the receiving call
// before `send` returns.
pendingReturns[msg.sender] = 0;

// msg.sender is not of type `address payable` and must be
// explicitly converted using `payable(msg.sender)` in order
// use the member function `send()`.

if (!payable(msg.sender).send(amount)) {
    // No need to call throw here, just reset the amount owing
    pendingReturns[msg.sender] = amount;
    return false;
}
return true;
}

/// End the auction and send the highest bid
/// to the beneficiary.
function auctionEnd() external {
    // It is a good guideline to structure functions that interact
    // with other contracts (i.e. they call functions or send Ether)
    // into three phases:
    // 1. checking conditions
    // 2. performing actions (potentially changing conditions)
    // 3. interacting with other contracts
    // If these phases are mixed up, the other contract could call
    // back into the current contract and modify the state or cause
    // effects (ether payout) to be performed multiple times.
    // If functions called internally include interaction with external
    // contracts, they also have to be considered interaction with
    // external contracts.

    // 1. Conditions
    if (block.timestamp < auctionEndTime)
        revert AuctionNotYetEnded();
    if (ended)
        revert AuctionEndAlreadyCalled();

    // 2. Effects
    ended = true;
    emit AuctionEnded(highestBidder, highestBid);

    // 3. Interaction
    beneficiary.transfer(highestBid);
}
}

```

Blind Auction

The previous open auction is extended to a blind auction in the following. The advantage of a blind auction is that there is no time pressure towards the end of the bidding period. Creating a blind auction on a transparent computing platform might sound like a contradiction, but cryptography comes to the rescue.

During the **bidding period**, a bidder does not actually send their bid, but only a hashed version of it. Since it is currently considered practically impossible to find two (sufficiently long) values whose hash values are equal, the bidder commits to the bid by that. After the end of the bidding period, the bidders have to reveal their bids: They send their values unencrypted and the contract checks that the hash value is the same as the one provided during the bidding period.

Another challenge is how to make the auction **binding and blind** at the same time: The only way to prevent the bidder from just not sending the money after they won the auction is to make them send it together with the bid. Since value transfers cannot be blinded in Ethereum, anyone can see the value.

The following contract solves this problem by accepting any value that is larger than the highest bid. Since this can of course only be checked during the reveal phase, some bids might be **invalid**, and this is on purpose (it even provides an explicit flag to place invalid bids with high value transfers): Bidders can confuse competition by placing several high or low invalid bids.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    // Errors that describe failures.

    /// The function has been called too early.
    /// Try again at `time`.
    error TooEarly(uint time);
    /// The function has been called too late.
    /// It cannot be called after `time`.
    error TooLate(uint time);
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    // Modifiers are a convenient way to validate inputs to
```

(continues on next page)

(continued from previous page)

```

// functions. `onlyBefore` is applied to `bid` below:
// The new function body is the modifier's body where
// `_` is replaced by the old function body.
modifier onlyBefore(uint time) {
    if (block.timestamp >= time) revert TooLate(time);
    _;
}
modifier onlyAfter(uint time) {
    if (block.timestamp <= time) revert TooEarly(time);
    _;
}

constructor(
    uint biddingTime,
    uint revealTime,
    address payable beneficiaryAddress
) {
    beneficiary = beneficiaryAddress;
    biddingEnd = block.timestamp + biddingTime;
    revealEnd = biddingEnd + revealTime;
}

/// Place a blinded bid with `blindedBid` =
/// keccak256(abi.encodePacked(value, fake, secret)).
/// The sent ether is only refunded if the bid is correctly
/// revealed in the revealing phase. The bid is valid if the
/// ether sent together with the bid is at least "value" and
/// "fake" is not true. Setting "fake" to true and sending
/// not the exact amount are ways to hide the real bid but
/// still make the required deposit. The same address can
/// place multiple bids.
function bid(bytes32 blindedBid)
    external
    payable
    onlyBefore(biddingEnd)
{
    bids[msg.sender].push(Bid({
        blindedBid: blindedBid,
        deposit: msg.value
    }));
}

/// Reveal your blinded bids. You will get a refund for all
/// correctly blinded invalid bids and for all bids except for
/// the totally highest.
function reveal(
    uint[] calldata values,
    bool[] calldata fakes,
    bytes32[] calldata secrets
)
    external
    onlyAfter(biddingEnd)

```

(continues on next page)

(continued from previous page)

```

    onlyBefore(revealEnd)
{
    uint length = bids[msg.sender].length;
    require(values.length == length);
    require(fakes.length == length);
    require(secrets.length == length);

    uint refund;
    for (uint i = 0; i < length; i++) {
        Bid storage bidToCheck = bids[msg.sender][i];
        (uint value, bool fake, bytes32 secret) =
            (values[i], fakes[i], secrets[i]);
        if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,
            secret))) {
            // Bid was not actually revealed.
            // Do not refund deposit.
            continue;
        }
        refund += bidToCheck.deposit;
        if (!fake && bidToCheck.deposit >= value) {
            if (placeBid(msg.sender, value))
                refund -= value;
        }
        // Make it impossible for the sender to re-claim
        // the same deposit.
        bidToCheck.blindedBid = bytes32(0);
    }
    payable(msg.sender).transfer(refund);
}

/// Withdraw a bid that was overbid.
function withdraw() external {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        // It is important to set this to zero because the recipient
        // can call this function again as part of the receiving call
        // before `transfer` returns (see the remark above about
        // conditions -> effects -> interaction).
        pendingReturns[msg.sender] = 0;

        payable(msg.sender).transfer(amount);
    }
}

/// End the auction and send the highest bid
/// to the beneficiary.
function auctionEnd()
    external
    onlyAfter(revealEnd)
{
    if (ended) revert AuctionEndAlreadyCalled();
    emit AuctionEnded(highestBidder, highestBid);
}

```

(continues on next page)

(continued from previous page)

```

ended = true;
beneficiary.transfer(highestBid);
}

// This is an "internal" function which means that it
// can only be called from the contract itself (or from
// derived contracts).
function placeBid(address bidder, uint value) internal
    returns (bool success)
{
    if (value <= highestBid) {
        return false;
    }
    if (highestBidder != address(0)) {
        // Refund the previously highest bidder.
        pendingReturns[highestBidder] += highestBid;
    }
    highestBid = value;
    highestBidder = bidder;
    return true;
}
}

```

3.3.3 Safe Remote Purchase

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment here: There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool condition_) {
        require(condition_);
    }
}

```

(continues on next page)

(continued from previous page)

```

        -;
    }

/// Only the buyer can call this function.
error OnlyBuyer();
/// Only the seller can call this function.
error OnlySeller();
/// The function cannot be called at the current state.
error InvalidState();
/// The provided value has to be even.
error ValueNotEven();

modifier onlyBuyer() {
    if (msg.sender != buyer)
        revert OnlyBuyer();
    -;
}

modifier onlySeller() {
    if (msg.sender != seller)
        revert OnlySeller();
    -;
}

modifier inState(State state_) {
    if (state != state_)
        revert InvalidState();
    -;
}

event Aborted();
event PurchaseConfirmed();
event ItemReceived();
event SellerRefunded();

// Ensure that `msg.value` is an even number.
// Division will truncate if it is an odd number.
// Check via multiplication that it wasn't an odd number.
constructor() payable {
    seller = payable(msg.sender);
    value = msg.value / 2;
    if ((2 * value) != msg.value)
        revert ValueNotEven();
}

/// Abort the purchase and reclaim the ether.
/// Can only be called by the seller before
/// the contract is locked.
function abort()
    external
    onlySeller
    inState(State.Created)

```

(continues on next page)

(continued from previous page)

```

{
    emit Aborted();
    state = State.Inactive;
    // We use transfer here directly. It is
    // reentrancy-safe, because it is the
    // last call in this function and we
    // already changed the state.
    seller.transfer(address(this).balance);
}

/// Confirm the purchase as buyer.
/// Transaction has to include `2 * value` ether.
/// The ether will be locked until confirmReceived
/// is called.
function confirmPurchase()
    external
    inState(State.Created)
    condition(msg.value == (2 * value))
    payable
{
    emit PurchaseConfirmed();
    buyer = payable(msg.sender);
    state = State.Locked;
}

/// Confirm that you (the buyer) received the item.
/// This will release the locked ether.
function confirmReceived()
    external
    onlyBuyer
    inState(State.Locked)
{
    emit ItemReceived();
    // It is important to change the state first because
    // otherwise, the contracts called using `send` below
    // can call in again here.
    state = State.Release;

    buyer.transfer(value);
}

/// This function refunds the seller, i.e.
/// pays back the locked funds of the seller.
function refundSeller()
    external
    onlySeller
    inState(State.Release)
{
    emit SellerRefunded();
    // It is important to change the state first because
    // otherwise, the contracts called using `send` below
    // can call in again here.
}

```

(continues on next page)

(continued from previous page)

```

state = State.Inactive;

seller.transfer(3 * value);
}
}

```

3.3.4 Micropayment Channel

In this section we will learn how to build an example implementation of a payment channel. It uses cryptographic signatures to make repeated transfers of Ether between the same parties secure, instantaneous, and without transaction fees. For the example, we need to understand how to sign and verify signatures, and setup the payment channel.

Creating and verifying signatures

Imagine Alice wants to send some Ether to Bob, i.e. Alice is the sender and Bob is the recipient.

Alice only needs to send cryptographically signed messages off-chain (e.g. via email) to Bob and it is similar to writing checks.

Alice and Bob use signatures to authorise transactions, which is possible with smart contracts on Ethereum. Alice will build a simple smart contract that lets her transmit Ether, but instead of calling a function herself to initiate a payment, she will let Bob do that, and therefore pay the transaction fee.

The contract will work as follows:

1. Alice deploys the `ReceiverPays` contract, attaching enough Ether to cover the payments that will be made.
2. Alice authorises a payment by signing a message with her private key.
3. Alice sends the cryptographically signed message to Bob. The message does not need to be kept secret (explained later), and the mechanism for sending it does not matter.
4. Bob claims his payment by presenting the signed message to the smart contract, it verifies the authenticity of the message and then releases the funds.

Creating the signature

Alice does not need to interact with the Ethereum network to sign the transaction, the process is completely offline. In this tutorial, we will sign messages in the browser using `web3.js` and `MetaMask`, using the method described in EIP-712, as it provides a number of other security benefits.

```

/// Hashing first makes things easier
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log("Signed
  ↵"); });

```

Note: The `web3.eth.personal.sign` prepends the length of the message to the signed data. Since we hash first, the message will always be exactly 32 bytes long, and thus this length prefix is always the same.

What to Sign

For a contract that fulfils payments, the signed message must include:

1. The recipient's address.
2. The amount to be transferred.
3. Protection against replay attacks.

A replay attack is when a signed message is reused to claim authorization for a second action. To avoid replay attacks we use the same technique as in Ethereum transactions themselves, a so-called nonce, which is the number of transactions sent by an account. The smart contract checks if a nonce is used multiple times.

Another type of replay attack can occur when the owner deploys a `ReceiverPays` smart contract, makes some payments, and then destroys the contract. Later, they decide to deploy the `RecipientPays` smart contract again, but the new contract does not know the nonces used in the previous deployment, so the attacker can use the old messages again.

Alice can protect against this attack by including the contract's address in the message, and only messages containing the contract's address itself will be accepted. You can find an example of this in the first two lines of the `claimPayment()` function of the full contract at the end of this section.

Packing arguments

Now that we have identified what information to include in the signed message, we are ready to put the message together, hash it, and sign it. For simplicity, we concatenate the data. The `ethereumjs-abi` library provides a function called `soliditySHA3` that mimics the behaviour of Solidity's `keccak256` function applied to arguments encoded using `abi.encodePacked`. Here is a JavaScript function that creates the proper signature for the `ReceiverPays` example:

```
// recipient is the address that should be paid.
// amount, in wei, specifies how much ether should be sent.
// nonce can be any unique number to prevent replay attacks
// contractAddress is used to prevent cross-contract replay attacks
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

Recovering the Message Signer in Solidity

In general, ECDSA signatures consist of two parameters, `r` and `s`. Signatures in Ethereum include a third parameter called `v`, that you can use to verify which account's private key was used to sign the message, and the transaction's sender. Solidity provides a built-in function `ecrecover` that accepts a message along with the `r`, `s` and `v` parameters and returns the address that was used to sign the message.

Extracting the Signature Parameters

Signatures produced by web3.js are the concatenation of r, s and v, so the first step is to split these parameters apart. You can do this on the client-side, but doing it inside the smart contract means you only need to send one signature parameter rather than three. Splitting apart a byte array into its constituent parts is a mess, so we use *inline assembly* to do the job in the `splitSignature` function (the third function in the full contract at the end of this section).

Computing the Message Hash

The smart contract needs to know exactly what parameters were signed, and so it must recreate the message from the parameters and use that for signature verification. The functions `prefixed` and `recoverSigner` do this in the `claimPayment` function.

The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature) ↵
    external {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce, this)));
    }

    require(recoverSigner(message, signature) == owner);

    payable(msg.sender).transfer(amount);
}

/// destroy the contract and reclaim the leftover funds.
function shutdown() external {
    require(msg.sender == owner);
    selfdestruct(payable(msg.sender));
}

/// signature methods.
function splitSignature(bytes memory sig)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
{
    require(sig.length == 65);
```

(continues on next page)

(continued from previous page)

```

assembly {
    // first 32 bytes, after the length prefix.
    r := mload(add(sig, 32))
    // second 32 bytes.
    s := mload(add(sig, 64))
    // final byte (first byte of the next 32 bytes).
    v := byte(0, mload(add(sig, 96)))
}

return (v, r, s);
}

function recoverSigner(bytes32 message, bytes memory sig)
internal
pure
returns (address)
{
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

    return ecrecover(message, v, r, s);
}

/// builds a prefixed hash to mimic the behavior of eth_sign.
function prefixed(bytes32 hash) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
}
}

```

Writing a Simple Payment Channel

Alice now builds a simple but complete implementation of a payment channel. Payment channels use cryptographic signatures to make repeated transfers of Ether securely, instantaneously, and without transaction fees.

What is a Payment Channel?

Payment channels allow participants to make repeated transfers of Ether without using transactions. This means that you can avoid the delays and fees associated with transactions. We are going to explore a simple unidirectional payment channel between two parties (Alice and Bob). It involves three steps:

1. Alice funds a smart contract with Ether. This “opens” the payment channel.
2. Alice signs messages that specify how much of that Ether is owed to the recipient. This step is repeated for each payment.
3. Bob “closes” the payment channel, withdrawing his portion of the Ether and sending the remainder back to the sender.

Note: Only steps 1 and 3 require Ethereum transactions, step 2 means that the sender transmits a cryptographically signed message to the recipient via off chain methods (e.g. email). This means only two transactions are required to support any number of transfers.

Bob is guaranteed to receive his funds because the smart contract escrows the Ether and honours a valid signed message. The smart contract also enforces a timeout, so Alice is guaranteed to eventually recover her funds even if the recipient refuses to close the channel. It is up to the participants in a payment channel to decide how long to keep it open. For a short-lived transaction, such as paying an internet café for each minute of network access, the payment channel may be kept open for a limited duration. On the other hand, for a recurring payment, such as paying an employee an hourly wage, the payment channel may be kept open for several months or years.

Opening the Payment Channel

To open the payment channel, Alice deploys the smart contract, attaching the Ether to be escrowed and specifying the intended recipient and a maximum duration for the channel to exist. This is the function `SimplePaymentChannel` in the contract, at the end of this section.

Making Payments

Alice makes payments by sending signed messages to Bob. This step is performed entirely outside of the Ethereum network. Messages are cryptographically signed by the sender and then transmitted directly to the recipient.

Each message includes the following information:

- The smart contract's address, used to prevent cross-contract replay attacks.
- The total amount of Ether that is owed the recipient so far.

A payment channel is closed just once, at the end of a series of transfers. Because of this, only one of the messages sent is redeemed. This is why each message specifies a cumulative total amount of Ether owed, rather than the amount of the individual micropayment. The recipient will naturally choose to redeem the most recent message because that is the one with the highest total. The nonce per-message is not needed anymore, because the smart contract only honours a single message. The address of the smart contract is still used to prevent a message intended for one payment channel from being used for a different channel.

Here is the modified JavaScript code to cryptographically sign a message from the previous section:

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.soliditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress is used to prevent cross-contract replay attacks.
// amount, in wei, specifies how much Ether should be sent.

function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
```

(continues on next page)

(continued from previous page)

```
    signMessage(message, callback);
}
```

Closing the Payment Channel

When Bob is ready to receive his funds, it is time to close the payment channel by calling a `close` function on the smart contract. Closing the channel pays the recipient the Ether they are owed and destroys the contract, sending any remaining Ether back to Alice. To close the channel, Bob needs to provide a message signed by Alice.

The smart contract must verify that the message contains a valid signature from the sender. The process for doing this verification is the same as the process the recipient uses. The Solidity functions `isValidSignature` and `recoverSigner` work just like their JavaScript counterparts in the previous section, with the latter function borrowed from the `ReceiverPays` contract.

Only the payment channel recipient can call the `close` function, who naturally passes the most recent payment message because that message carries the highest total owed. If the sender were allowed to call this function, they could provide a message with a lower amount and cheat the recipient out of what they are owed.

The function verifies the signed message matches the given parameters. If everything checks out, the recipient is sent their portion of the Ether, and the sender is sent the rest via a `selfdestruct`. You can see the `close` function in the full contract.

Channel Expiration

Bob can close the payment channel at any time, but if they fail to do so, Alice needs a way to recover her escrowed funds. An *expiration* time was set at the time of contract deployment. Once that time is reached, Alice can call `claimTimeout` to recover her funds. You can see the `claimTimeout` function in the full contract.

After this function is called, Bob can no longer receive any Ether, so it is important that Bob closes the channel before the expiration is reached.

The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract SimplePaymentChannel {
    address payable public sender;           // The account sending payments.
    address payable public recipient;        // The account receiving the payments.
    uint256 public expiration;               // Timeout in case the recipient never closes.

    constructor (address payable recipientAddress, uint256 duration)
        payable
    {
        sender = payable(msg.sender);
        recipient = recipientAddress;
        expiration = block.timestamp + duration;
    }

    /// the recipient can close the channel at any time by presenting a
    /// signed amount from the sender. the recipient will be sent that amount,
}
```

(continues on next page)

(continued from previous page)

```

/// and the remainder will go back to the sender
function close(uint256 amount, bytes memory signature) external {
    require(msg.sender == recipient);
    require(isValidSignature(amount, signature));

    recipient.transfer(amount);
    selfdestruct(sender);
}

/// the sender can extend the expiration at any time
function extend(uint256 newExpiration) external {
    require(msg.sender == sender);
    require(newExpiration > expiration);

    expiration = newExpiration;
}

/// if the timeout is reached without the recipient closing the channel,
/// then the Ether is released back to the sender.
function claimTimeout() external {
    require(block.timestamp >= expiration);
    selfdestruct(sender);
}

function isValidSignature(uint256 amount, bytes memory signature)
    internal
    view
    returns (bool)
{
    bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

    // check that the signature is from the payment sender
    return recoverSigner(message, signature) == sender;
}

/// All functions below this are just taken from the chapter
/// 'creating and verifying signatures' chapter.

function splitSignature(bytes memory sig)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
{
    require(sig.length == 65);

    assembly {
        // first 32 bytes, after the length prefix
        r := mload(add(sig, 32))
        // second 32 bytes
        s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes)
        v := byte(0, mload(add(sig, 96)))
    }
}

```

(continues on next page)

(continued from previous page)

```

        }

        return (v, r, s);
    }

function recoverSigner(bytes32 message, bytes memory sig)
    internal
    pure
    returns (address)
{
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

    return ecrecover(message, v, r, s);
}

/// builds a prefixed hash to mimic the behavior of eth_sign.
function prefixed(bytes32 hash) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
}
}

```

Note: The function `splitSignature` does not use all security checks. A real implementation should use a more rigorously tested library, such as openzeppelin's [version](#) of this code.

Verifying Payments

Unlike in the previous section, messages in a payment channel aren't redeemed right away. The recipient keeps track of the latest message and redeems it when it's time to close the payment channel. This means it's critical that the recipient perform their own verification of each message. Otherwise there is no guarantee that the recipient will be able to get paid in the end.

The recipient should verify each message using the following process:

1. Verify that the contract address in the message matches the payment channel.
2. Verify that the new total is the expected amount.
3. Verify that the new total does not exceed the amount of Ether escrowed.
4. Verify that the signature is valid and comes from the payment channel sender.

We'll use the `ethereumjs-util` library to write this verification. The final step can be done a number of ways, and we use JavaScript. The following code borrows the `constructPaymentMessage` function from the signing [JavaScript code](#) above:

```

// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

```

(continues on next page)

(continued from previous page)

```

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}

```

3.3.5 Modular Contracts

A modular approach to building your contracts helps you reduce the complexity and improve the readability which will help to identify bugs and vulnerabilities during development and code review. If you specify and control the behaviour of each module in isolation, the interactions you have to consider are only those between the module specifications and not every other moving part of the contract. In the example below, the contract uses the `move` method of the `Balances library` to check that balances sent between addresses match what you expect. In this way, the `Balances` library provides an isolated component that properly tracks balances of accounts. It is easy to verify that the `Balances` library never produces negative balances or overflows and the sum of all balances is an invariant across the lifetime of the contract.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address to,
    → uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) external returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;
    }
}

```

(continues on next page)

(continued from previous page)

```

}

function transferFrom(address from, address to, uint amount) external returns (bool success) {
    require(allowed[from][msg.sender] >= amount);
    allowed[from][msg.sender] -= amount;
    balances.move(from, to, amount);
    emit Transfer(from, to, amount);
    return true;
}

function approve(address spender, uint tokens) external returns (bool success) {
    require(allowed[msg.sender][spender] == 0, "");
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
}

function balanceOf(address tokenOwner) external view returns (uint balance) {
    return balances[tokenOwner];
}
}

```

3.4 Layout of a Solidity Source File

Source files can contain an arbitrary number of *contract definitions*, *import*, *pragma* and *using for* directives and *struct*, *enum*, *function*, *error* and *constant variable* definitions.

3.4.1 SPDX License Identifier

Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable **SPDX license identifiers**. Every source file should start with a comment indicating its license:

```
// SPDX-License-Identifier: MIT
```

The compiler does not validate that the license is part of the [list allowed by SPDX](#), but it does include the supplied string in the *bytecode metadata*.

If you do not want to specify a license or if the source code is not open-source, please use the special value UNLICENSED. Note that UNLICENSED (no usage allowed, not present in SPDX license list) is different from UNLICENSE (grants all rights to everyone). Solidity follows [the npm recommendation](#).

Supplying this comment of course does not free you from other obligations related to licensing like having to mention a specific license header in each source file or the original copyright holder.

The comment is recognized by the compiler anywhere in the file at the file level, but it is recommended to put it at the top of the file.

More information about how to use SPDX license identifiers can be found at the [SPDX website](#).

3.4.2 Pragmas

The `pragma` keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project. If you `import` another file, the pragma from that file does *not* automatically apply to the importing file.

Version Pragma

Source files can (and should) be annotated with a version pragma to reject compilation with future compiler versions that might introduce incompatible changes. We try to keep these to an absolute minimum and introduce them in a way that changes in semantics also require changes in the syntax, but this is not always possible. Because of this, it is always a good idea to read through the changelog at least for releases that contain breaking changes. These releases always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows: `pragma solidity ^0.5.2;`

A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0 (this second condition is added by using `^`). Because there will be no breaking changes until version `0.6.0`, you can be sure that your code compiles the way you intended. The exact version of the compiler is not fixed, so that bugfix releases are still possible.

It is possible to specify more complex rules for the compiler version, these follow the same syntax used by `npm`.

Note: Using the version pragma *does not* change the version of the compiler. It also *does not* enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.

ABI Coder Pragma

By using `pragma abicoder v1` or `pragma abicoder v2` you can select between the two implementations of the ABI encoder and decoder.

The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. It might produce less optimal code and has not received as much testing as the old encoder, but is considered non-experimental as of Solidity 0.6.0. You still have to explicitly activate it using `pragma abicoder v2;`. Since it will be activated by default starting from Solidity 0.8.0, there is the option to select the old coder using `pragma abicoder v1;`.

The set of types supported by the new encoder is a strict superset of the ones supported by the old one. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-`abicoder v2` contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling `abicoder v2` for your contract is enough to make the error go away.

Note: This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.

Note: Up to Solidity 0.7.4, it was possible to select the ABI coder v2 by using `pragma experimental ABIEncoderV2`, but it was not possible to explicitly select coder v1 because it was the default.

Experimental Pragma

The second pragma is the experimental pragma. It can be used to enable features of the compiler or language that are not yet enabled by default. The following experimental pragmas are currently supported:

ABIEncoderV2

Because the ABI coder v2 is not considered experimental anymore, it can be selected via `pragma abicoder v2` (please see above) since Solidity 0.7.4.

SMTChecker

This component has to be enabled when the Solidity compiler is built and therefore it is not available in all Solidity binaries. The *build instructions* explain how to activate this option. It is activated for the Ubuntu PPA releases in most versions, but not for the Docker images, Windows binaries or the statically-built Linux binaries. It can be activated for solc-js via the `smtCallback` if you have an SMT solver installed locally and run solc-js via node (not via the browser).

If you use `pragma experimental SMTChecker;`, then you get additional *safety warnings* which are obtained by querying an SMT solver. The component does not yet support all features of the Solidity language and likely outputs many warnings. In case it reports unsupported features, the analysis may not be fully sound.

3.4.3 Importing other Source Files

Syntax and Semantics

Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on). However, Solidity does not support the concept of a `default export`.

At a global level, you can use import statements of the following form:

```
import "filename";
```

The `filename` part is called an *import path*. This statement imports all global symbols from “`filename`” (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity). This form is not recommended for use, because it unpredictably pollutes the namespace. If you add new top-level items inside “`filename`”, they automatically appear in all files that import like this from “`filename`”. It is better to import specific symbols explicitly.

The following example creates a new global symbol `symbolName` whose members are all the global symbols from “`filename`”:

```
import * as symbolName from "filename";
```

which results in all global symbols being available in the format `symbolName.symbol`.

A variant of this syntax that is not part of ES6, but possibly useful is:

```
import "filename" as symbolName;
```

which is equivalent to `import * as symbolName from "filename";`.

If there is a naming collision, you can rename symbols while importing. For example, the code below creates new global symbols `alias` and `symbol2` which reference `symbol1` and `symbol2` from inside “`filename`”, respectively.

```
import {symbol1 as alias, symbol2} from "filename";
```

Import Paths

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. For this reason import paths do not refer directly to files in the host filesystem. Instead the compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. The import path specified in an import statement is translated into a source unit name and used to find the corresponding source unit in this database.

Using the [Standard JSON](#) API it is possible to directly provide the names and content of all the source files as a part of the compiler input. In this case source unit names are truly arbitrary. If, however, you want the compiler to automatically find and load source code into the VFS, your source unit names need to be structured in a way that makes it possible for an [import callback](#) to locate them. When using the command-line compiler the default import callback supports only loading source code from the host filesystem, which means that your source unit names must be paths. Some environments provide custom callbacks that are more versatile. For example the [Remix IDE](#) provides one that lets you import files from [HTTP](#), [IPFS](#) and [Swarm URLs](#) or refer directly to packages in NPM registry.

For a complete description of the virtual filesystem and the path resolution logic used by the compiler see [Path Resolution](#).

3.4.4 Comments

Single-line comments (//) and multi-line comments /*...*/ are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Note: A single-line comment is terminated by any unicode line terminator (LF, VF, FF, CR, NEL, LS or PS) in UTF-8 encoding. The terminator is still part of the source code after the comment, so if it is not an ASCII symbol (these are NEL, LS and PS), it will lead to a parser error.

Additionally, there is another type of comment called a NatSpec comment, which is detailed in the [style guide](#). They are written with a triple slash (///) or a double asterisk block (/** ... */) and they should be used directly above function declarations or statements.

3.5 Structure of a Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of [State Variables](#), [Functions](#), [Function Modifiers](#), [Events](#), [Errors](#), [Struct Types](#) and [Enum Types](#). Furthermore, contracts can inherit from other contracts.

There are also special kinds of contracts called [libraries](#) and [interfaces](#).

The section about [contracts](#) contains more details than this section, which serves to provide a quick overview.

3.5.1 State Variables

State variables are variables whose values are permanently stored in contract storage.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

See the [Types](#) section for valid state variable types and [Visibility and Getters](#) for possible choices for visibility.

3.5.2 Functions

Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract SimpleAuction {
    function bid() public payable { // Function
        // ...
    }
}

// Helper function defined outside of a contract
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Function Calls can happen internally or externally and have different levels of *visibility* towards other contracts. *Functions* accept *parameters and return variables* to pass parameters and values between them.

3.5.3 Function Modifiers

Function modifiers can be used to amend the semantics of functions in a declarative way (see [Function Modifiers](#) in the contracts section).

Overloading, that is, having the same modifier name with different parameters, is not possible.

Like functions, modifiers can be *overridden*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only the seller can call this function"
        );
        _;
    }
}
```

(continues on next page)

(continued from previous page)

```

    msg.sender == seller,
    "Only seller can call this."
);
-
}

function abort() public view onlySeller { // Modifier usage
    //
}
}

```

3.5.4 Events

Events are convenience interfaces with the EVM logging facilities.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() public payable {
        //
        emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}

```

See [Events](#) in contracts section for information on how events are declared and can be used from within a dapp.

3.5.5 Errors

Errors allow you to define descriptive names and data for failure situations. Errors can be used in [revert statements](#). In comparison to string descriptions, errors are much cheaper and allow you to encode additional data. You can use NatSpec to describe the error to the user.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Not enough funds for transfer. Requested `requested`,
/// but only `available` available.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}

```

(continues on next page)

(continued from previous page)

```
// ...
}
```

See [Errors and the Revert Statement](#) in the contracts section for more information.

3.5.6 Struct Types

Structs are custom defined types that can group several variables (see [Structs](#) in types section).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

3.5.7 Enum Types

Enums can be used to create custom types with a finite set of ‘constant values’ (see [Enums](#) in types section).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

3.6 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see [Order of Precedence of Operators](#).

The concept of “undefined” or “null” values does not exist in Solidity, but newly declared variables always have a *default value* dependent on its type. To handle any unexpected values, you should use the [revert function](#) to revert the whole transaction, or return a tuple with a second `bool` value denoting success.

3.6.1 Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- `!` (logical negation)
- `&&` (logical conjunction, “and”)
- `||` (logical disjunction, “or”)
- `==` (equality)
- `!=` (inequality)

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int / uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of 8 (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: `<<` (left shift), `>>` (right shift)
- Arithmetic operators: `+`, `-`, unary `-` (only for signed integers), `*`, `/`, `%` (modulo), `**` (exponentiation)

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

Warning: Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to `2**32 - 1`. There are two modes in which arithmetic is performed on these types: The “wrapping” or “unchecked” mode and the “checked” mode. By default, arithmetic is always “checked”, which means that if the result of an operation falls outside the value range of the type, the call is reverted through a *failing assertion*. You can switch to “unchecked” mode using `unchecked { ... }`. More details can be found in the section about [unchecked](#).

Comparisons

The value of a comparison is the one obtained by comparing the integer value.

Bit operations

Bit operations are performed on the two's complement representation of the number. This means that, for example `~int256(0) == int256(-1)`.

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by a signed type will produce a compilation error.

Shifts can be “simulated” using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

- `x << y` is equivalent to the mathematical expression `x * 2**y`.
- `x >> y` is equivalent to the mathematical expression `x / 2**y`, rounded towards negative infinity.

Warning: Before version 0.5.0 a right shift `x >> y` for negative `x` was equivalent to the mathematical expression `x / 2**y` rounded towards zero, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

Note: Overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated.

Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow:

By default, all arithmetic is checked for under- or overflow, but this can be disabled using the `unchecked block`, resulting in wrapping arithmetic. More details can be found in that section.

The expression `-x` is equivalent to `(T(0) - x)` where `T` is the type of `x`. It can only be applied to signed types. The value of `-x` can be positive if `x` is negative. There is another caveat also resulting from two's complement representation:

If you have `int x = type(int).min;`, then `-x` does not fit the positive range. This means that `unchecked { assert(-x == x); }` works, and the expression `-x` when used in checked mode will result in a failing assertion.

Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This means that `int256(-5) / int256(2) == int256(-2)`.

Note that in contrast, division on *literals* results in fractional values of arbitrary precision.

Note: Division by zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Note: The expression `type(int).min / (-1)` is the only case where division causes an overflow. In checked arithmetic mode, this will cause a failing assertion, while in wrapping mode, the value will be `type(int).min`.

Modulo

The modulo operation `a % n` yields the remainder `r` after the division of the operand `a` by the operand `n`, where `q = int(a / n)` and `r = a - (n * q)`. This means that modulo results in the same sign as its left operand (or zero) and `a % n == -(-a % n)` holds for negative `a`:

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

Note: Modulo with zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behaviour.

Note: In checked mode, exponentiation only uses the comparatively cheap `exp` opcode for small bases. For the cases of `x**3`, the expression `x*x*x` might be cheaper. In any case, gas cost tests and the use of the optimizer are advisable.

Note: Note that `0**0` is defined by the EVM as 1.

Fixed Point Numbers

Warning: Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

fixed / ufixed: Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represents the number of bits taken by the type and `N` represents how many decimal points are available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x18` and `fixed128x18`, respectively.

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Arithmetic operators: `+`, `-`, unary `-`, `*`, `/`, `%` (modulo)

Note: The main difference between floating point (`float` and `double` in many languages, more precisely IEEE 754 numbers) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is.

Address

The address type comes in two flavours, which are largely identical:

- `address`: Holds a 20 byte value (size of an Ethereum address).
- `address payable`: Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while you are not supposed to send Ether to a plain `address`, for example because it might be a smart contract that was not built to accept Ether.

Type conversions:

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`.

Explicit conversions to and from `address` are allowed for `uint160`, integer literals, `bytes20` and contract types.

Only expressions of type `address` and contract-type can be converted to the type `address payable` via the explicit conversion `payable(...)`. For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a `receive` or a payable fallback function. Note that `payable(0)` is valid and is an exception to this rule.

Note: If you need a variable of type `address` and plan to send Ether to it, then declare its type as `address payable` to make this requirement visible. Also, try to make this distinction or conversion as early as possible.

Operators:

- `<=`, `<`, `==`, `!=`, `>=` and `>`

Warning: If you convert a type that uses a larger byte size to an address, for example `bytes32`, then the address is truncated. To reduce conversion ambiguity version 0.4.24 and higher of the compiler force you make the truncation explicit in the conversion. Take for example the 32-byte value `0x11112222333344445555666677778889999AAAABBBCCCCDDDEEEFFFFCCCC`.

You can use `address(uint160(bytes20(b)))`, which results in `0x11112222333344445555666677778889999aAaa`, or you can use `address(uint160(uint256(b)))`, which results in `0x77778889999AaAAbBbbCcccddDdeeeEfFFfCcCc`.

Note: The distinction between `address` and `address payable` was introduced with version 0.5.0. Also starting from that version, contracts do not derive from the `address` type, but can still be explicitly converted to `address` or to `address payable`, if they have a receive or payable fallback function.

Members of Addresses

For a quick reference of all members of `address`, see [Members of Address Types](#).

- `balance` and `transfer`

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to a payable address using the `transfer` function:

```
address payable x = payable(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure.

Note: If `x` is a contract address, its code (more specifically: its [Receive Ether Function](#), if present, or otherwise its [Fallback Function](#), if present) will be executed together with the `transfer` call (this is a feature of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

- `send`

`Send` is the low-level counterpart of `transfer`. If the execution fails, the current contract will not stop with an exception, but `send` will return `false`.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: use a pattern where the recipient withdraws the money.

- `call`, `delegatecall` and `staticcall`

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

Warning: All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns. The regular way to interact with other contracts is to call a function on a contract object (`x.f()`).

Note: Previous versions of Solidity allowed these functions to receive arbitrary arguments and would also handle a first argument of type `bytes4` differently. These edge cases were removed in version 0.5.0.

It is possible to adjust the supplied gas with the `gas` modifier:

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature("register(string)", "MyName
˓→"));
```

Similarly, the supplied Ether value can be controlled too:

```
address(nameReg).call{value: 1 ether}(abi.encodeWithSignature("register(string)", "MyName
˓→));
```

Lastly, these modifiers can be combined. Their order does not matter:

```
address(nameReg).call{gas: 1000000, value: 1 ether}(abi.encodeWithSignature(
˓→"register(string)", "MyName"));
```

In a similar way, the function `delegatecall` can be used: the difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for `delegatecall` to be used.

Note: Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values. This function was removed in version 0.5.0.

Since byzantium `staticcall` can be used as well. This is basically the same as `call`, but will revert if the called function modifies the state in any way.

All three functions `call`, `delegatecall` and `staticcall` are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

The `gas` option is available on all three methods, while the `value` option is only available on `call`.

Note: It is best to avoid relying on hardcoded gas values in your smart contract code, regardless of whether state is read from or written to, as this can have many pitfalls. Also, access to gas might change in the future.

- `code` and `codehash`

You can query the deployed code for any smart contract. Use `.code` to get the EVM bytecode as a `bytes memory`, which might be empty. Use `.codehash` get the Keccak-256 hash of that code (as a `bytes32`). Note that `addr.codehash` is cheaper than using `keccak256(addr.code)`.

Note: All contracts can be converted to address type, so it is possible to query the balance of the current contract using `address(this).balance`.

Contract Types

Every `contract` defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the `address` type.

Explicit conversion to and from the `address payable` type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using `address(x)`. If the contract type does not have a receive or payable fallback function, the conversion to `address payable` can be done using `payable(address(x))`. You can find more information in the section about the [address type](#).

Note: Before version 0.5.0, contracts directly derived from the `address` type and there was no distinction between `address` and `address payable`.

If you declare a local variable of contract type (`MyContract c`), you can call functions on that contract. Take care to assign it from somewhere that is the same contract type.

You can also instantiate contracts (which means they are newly created). You can find more details in the '[Contracts via new](#)' section.

The data representation of a contract is identical to that of the `address` type and this type is also used in the [ABI](#).

Contracts do not support any operators.

The members of contract types are the external functions of the contract including any state variables marked as `public`.

For a contract C you can use `type(C)` to access [type information](#) about the contract.

Fixed-size byte arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32.

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: `<<` (left shift), `>>` (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

Members:

- `.length` yields the fixed length of the byte array (read-only).

Note: The type `bytes1[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

Note: Prior to version 0.8.0, `byte` used to be an alias for `bytes1`.

Dynamically-sized byte array

`bytes`:

Dynamically-sized byte array, see [Arrays](#). Not a value-type!

`string`:

Dynamically-sized UTF-8-encoded string, see [Arrays](#). Not a value-type!

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

Note: The mixed-case address checksum format is defined in [EIP-55](#).

Rational and Integer Literals

Integer literals are formed from a sequence of digits in the range 0-9. They are interpreted as decimals. For example, `69` means sixty nine. Octal literals do not exist in Solidity and leading zeros are invalid.

Decimal fractional literals are formed by a `.` with at least one number on one side. Examples include `1.`, `.1` and `1.3`.

Scientific notation in the form of `2e10` is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal `M` is equivalent to `M * 10**E`. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with anything other than a number literal expression (like boolean literals) or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, `(2**800 + 1) - 2**800` results in the constant 1 (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, `.5 * 8` results in the integer 4 (although non-integers were used in between).

Warning: While most operators produce a literal expression when applied to literals, there are certain operators that do not follow this pattern:

- Ternary operator `(... ? ... : ...)`,
- Array subscript `(<array>[<index>])`.

You might expect expressions like `255 + (true ? 1 : 0)` or `255 + [1, 2, 3][0]` to be equivalent to using the literal `256` directly, but in fact they are computed within the type `uint8` and can overflow.

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Shifts and exponentiation with literal numbers as left (or base) operand and integer types as the right (exponent) operand are always performed in the `uint256` (for non-negative literals) or `int256` (for a negative literals) type, regardless of the type of the right (exponent) operand.

Warning: Division on integer literals used to truncate in Solidity prior to version 0.4.0, but it now converts into a rational number, i.e. `5 / 2` is not equal to 2, but to 2.5.

Note: Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions `1 + 2` and `2 + 1` both belong to the same number literal type for the rational number three.

Note: Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Disregarding types, the value of the expression assigned to `b` below evaluates to an integer. Because `a` is of type `uint128`, the expression `2.5 + a` has to have a proper type, though. Since there is no common type for the type of `2.5` and `uint128`, the Solidity compiler does not accept this code.

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

String Literals and Types

String literals are written with either double or single-quotes ("foo" or 'bar'), and they can also be split into multiple consecutive parts ("foo" "bar" is equivalent to "foobar") which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; "foo" represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

For example, with `bytes32 samevar = "stringliteral"` the string literal is interpreted in its raw byte form when assigned to a `bytes32` type.

String literals can only contain printable ASCII characters, which means the characters between and including `0x20 .. 0x7E`.

Additionally, string literals also support the following escape characters:

- `\<newline>` (escapes an actual newline)
- `\\\` (backslash)
- `\'` (single quote)
- `\\"` (double quote)
- `\n` (newline)

- \r (carriage return)
- \t (tab)
- \xNN (hex escape, see below)
- \uNNNN (unicode escape, see below)

\xNN takes a hex value and inserts the appropriate byte, while \uNNNN takes a Unicode codepoint and inserts an UTF-8 sequence.

Note: Until version 0.8.0 there were three additional escape sequences: \b, \f and \v. They are commonly available in other languages but rarely needed in practice. If you do need them, they can still be inserted via hexadecimal escapes, i.e. \x08, \x0c and \x0b, respectively, just as any other ASCII character.

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote a backslash character and then (without separator) the character sequence abcdef.

```
"\n\"'\\abc"
def"
```

Any Unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a \.

Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword `unicode` – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

```
string memory a = unicode"Hello ";
```

Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`, `hex'0011_22_FF'`). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: `hex"00112233" hex"44556677"` is equivalent to `hex"0011223344556677"`

Hexadecimal literals behave like *string literals* and have the same convertibility restrictions.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a *Panic error* otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

The data representation is the same as for enums in C: The options are represented by subsequent unsigned integer values starting from 0.

Using `type(NameOfEnum).min` and `type(NameOfEnum).max` you can get the smallest and respectively largest value of the given enum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }

    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

Note: Enums can also be declared on the file level, outside of contract or library definitions.

User Defined Value Types

A user defined value type allows creating a zero cost abstraction over an elementary value type. This is similar to an alias, but with stricter type requirements.

A user defined value type is defined using `type C is V`, where C is the name of the newly introduced type and V has to be a built-in value type (the “underlying type”). The function `C.wrap` is used to convert from the underlying type to the custom type. Similarly, the function `C.unwrap` is used to convert from the custom type to the underlying type.

The type C does not have any operators or bound member functions. In particular, even the operator `==` is not defined. Explicit and implicit conversions to and from other types are disallowed.

The data-representation of values of such types are inherited from the underlying type and the underlying type is also used in the ABI.

The following example illustrates a custom type `UFixed256x18` representing a decimal fixed point type with 18 decimals and a minimal library to do arithmetic operations on the type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user defined value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
    uint constant multiplier = 10**18;

    /// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }
    /// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// Take the floor of a UFixed256x18 number.
    /// @return the largest integer that does not exceed `a`.
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// Turns a uint256 into a UFixed256x18 of the same value.
    /// Reverts if the integer is too large.
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

Notice how `UFixed256x18.wrap` and `FixedMath.toUFixed256x18` have the same signature but perform two very different operations: The `UFixed256x18.wrap` function returns a `UFixed256x18` that has the same data representation as the input, whereas `toUFixed256x18` returns a `UFixed256x18` that has the same numerical value.

Function Types

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions:

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:

```
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return types>)]
```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

Conversions:

A function type A is implicitly convertible to a function type B if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of A is more restrictive than the state mutability of B. In particular:

- pure functions can be converted to view and non-payable functions
- view functions can be converted to non-payable functions
- payable functions can be converted to non-payable functions

No other conversions between function types are possible.

The rule about payable and non-payable might be a little confusing, but in essence, if a function is payable, this means that it also accepts a payment of zero Ether, so it also is non-payable. On the other hand, a non-payable function will reject Ether sent to it, so non-payable functions cannot be converted to payable functions.

If a function type variable is not initialised, calling it results in a [Panic error](#). The same happens if you call a function after using `delete` on it.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

A function of an internal type can be assigned to a variable of an internal function type regardless of where it is defined. This includes private, internal and public functions of both contracts and libraries as well as free functions. External function types, on the other hand, are only compatible with public and external contract functions. Libraries are excluded because they require a `delegatecall` and use [a different ABI convention for their selectors](#). Functions declared in interfaces do not have definitions so pointing at them does not make sense either.

Members:

External (or public) functions have the following members:

- `.address` returns the address of the contract of the function.
- `.selector` returns the [ABI function selector](#)

Note: External (or public) functions used to have the additional members `.gas(uint)` and `.value(uint)`. These were deprecated in Solidity 0.6.2 and removed in Solidity 0.7.0. Instead use `{gas: ...}` and `{value: ...}` to specify the amount of gas or the amount of wei sent to a function, respectively. See [External Function Calls](#) for more information.

Example that shows how to use the members:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.4 <0.9.0;

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
```

(continues on next page)

(continued from previous page)

```

    return this.f.selector;
}

function g() public {
    this.f{gas: 10, value: 800}();
}
}

```

Example that shows how to use internal function types:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library ArrayUtils {
    // internal functions can be used in internal library functions because
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }

    function range(uint length) internal pure returns (uint[] memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}

contract Pyramid {
    using ArrayUtils for *;
}

```

(continues on next page)

(continued from previous page)

```

function pyramid(uint l) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
}

function square(uint x) internal pure returns (uint) {
    return x * x;
}

function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
}
}

```

Another example that uses external function types:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}

contract OracleUser {
    Oracle constant private ORACLE_CONST = 
    ←Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(

```

(continues on next page)

(continued from previous page)

```

    msg.sender == address(ORACLE_CONST),
    "Only oracle can call this."
);
exchangeRate = response;
}
}

```

Note: Lambda or inline functions are planned but not yet supported.

3.6.2 Reference Types

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: `memory` (whose lifetime is limited to an external function call), `storage` (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or `calldata` (special data location that contains the function arguments).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

Data location

Every reference type has an additional annotation, the “data location”, about where it is stored. There are three data locations: `memory`, `storage` and `calldata`. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

Note: If you can, try to use `calldata` as data location because it will avoid copies and also makes sure that the data cannot be modified. Arrays and structs with `calldata` data location can also be returned from functions, but it is not possible to allocate such types.

Note: Prior to version 0.6.9 data location for reference-type arguments was limited to `calldata` in external functions, `memory` in public functions and either `memory` or `storage` in internal and private ones. Now `memory` and `calldata` are allowed in all functions regardless of their visibility.

Note: Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

Data location and assignment behaviour

Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a `local` storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

For example, an array of 5 dynamic arrays of `uint` is written as `uint[][][5]`. The notation is reversed compared to some other languages. In Solidity, `X[3]` is always an array containing three elements of type `X`, even if `X` is itself an array. This is not the case in other languages such as C.

Indices are zero-based, and access is in the opposite direction of the declaration.

For example, if you have a variable `uint[][] memory x`, you access the seventh `uint` in the third dynamic array using `x[2][6]`, and to access the third dynamic array, use `x[2]`. Again, if you have an array `T[5] a` for a type `T` that can also be an array, then `a[2]` always has type `T`.

Array elements can be of any type, including mapping or struct. The general restrictions for types apply, in that mappings can only be stored in the storage data location and publicly-visible functions need parameters that are [ABI types](#).

It is possible to mark state variable arrays `public` and have Solidity create a [getter](#). The numeric index becomes a required parameter for the getter.

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of the array, where `.push()` appends a zero-initialized element and returns a reference to it.

bytes and string as Arrays

Variables of type `bytes` and `string` are special arrays. The `bytes` type is similar to `bytes1[]`, but it is packed tightly in calldata and memory. `string` is equal to `bytes` but does not allow length or index access.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `string.concat(s1, s2)`.

You should use `bytes` over `bytes1[]` because it is cheaper, since using `bytes1[]` in memory adds 31 padding bytes between the elements. Note that in storage, the padding is absent due to tight packing, see [bytes and string](#). As a general rule, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

Note: If you want to access the byte-representation of a string `s`, use `bytes(s).length / bytes(s)[7] = 'x'`. Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual characters.

The functions `bytes.concat` and `string.concat`

You can concatenate an arbitrary number of `string` values using `string.concat`. The function returns a single `string` memory array that contains the contents of the arguments without padding. If you want to use parameters of other types that are not implicitly convertible to `string`, you need to convert them to `string` first.

Analogously, the `bytes.concat` function can concatenate an arbitrary number of `bytes` or `bytes1 ... bytes32` values. The function returns a single `bytes` memory array that contains the contents of the arguments without padding. If you want to use `string` parameters or other types that are not implicitly convertible to `bytes`, you need to convert them to `bytes` or `bytes1.../bytes32` first.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

contract C {
    string s = "Storage";
    function f(bytes calldata bc, string memory sm, bytes16 b) public view {
        string memory concatString = string.concat(s, string(bc), "Literal", sm);
        assert((bytes(s).length + bc.length + 7 + bytes(sm).length) ==_
            bytes(concatString).length);
    }
}
```

(continues on next page)

(continued from previous page)

```

    bytes memory concatBytes = bytes.concat(bytes(s), bc, bc[:2], "Literal",_
→bytes(sm), b);
    assert((bytes(s).length + bc.length + 2 + 7 + bytes(sm).length + b.length) ==_
→concatBytes.length);
}
}

```

If you call `string.concat` or `bytes.concat` without arguments they return an empty array.

Allocating Memory Arrays

Memory arrays with dynamic length can be created using the `new` operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the `.push` member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the *default value*.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}

```

Array Literals

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`). For example `[1, a, f(3)]`. The type of the array literal is determined as follows:

It is always a statically-sized memory array whose length is the number of expressions.

The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.

It is not enough that there is a type all the elements can be converted to. One of the elements has to be of that type.

In the example below, the type of `[1, 2, 3]` is `uint8[3]` memory, because the type of each of these constants is `uint8`. If you want the result to be a `uint[3]` memory type, you need to convert the first element to `uint`.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
}

```

(continues on next page)

(continued from previous page)

```
function g(uint[3] memory) public pure {
    // ...
}
```

The array literal [1, -1] is invalid because the type of the first expression is uint8 while the type of the second is int8 and they cannot be implicitly converted to each other. To make it work, you can use [int8(1), -1], for example.

Since fixed-size memory arrays of different type cannot be converted into each other (even if the base types can), you always have to specify a common base type explicitly if you want to use two-dimensional array literals:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffffff, 2], [uint24(0xff), 3],_
        [uint24(0xffff), 4]];
        // The following does not work, because some of the inner arrays are not of the_
        // right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

// This will not compile.
contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

It is planned to remove this restriction in the future, but it creates some complications because of how arrays are passed in the ABI.

If you want to initialize dynamically-sized arrays, you have to assign the individual elements:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}
```

Array Members

length:

Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push():

Dynamic storage arrays and `bytes` (not `string`) have a member function called `push()` that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like `x.push().t = 2` or `x.push() = b`.

push(x):

Dynamic storage arrays and `bytes` (not `string`) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

pop():

Dynamic storage arrays and `bytes` (not `string`) have a member function called `pop()` that you can use to remove an element from the end of the array. This also implicitly calls `delete` on the removed element. The function returns nothing.

Note: Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the “size” of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling `delete` on them.

Note: To use arrays of arrays in external (instead of public) functions, you need to activate ABI coder v2.

Note: In EVM versions before Byzantium, it was not possible to access dynamic arrays return from function calls. If you call functions that return dynamic arrays, make sure to use an EVM that is set to Byzantium mode.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ArrayContract {
    uint[2**20] aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    // Because of that, T[] is always a dynamic array of T, even if T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] pairsOfFlags;

    // newPairs is stored in memory - the only possibility
    // for public contract function arguments
    function setAllFlagPairs(bool[2][] memory newPairs) public {

```

(continues on next page)

(continued from previous page)

```

// assignment to a storage array performs a copy of ``newPairs`` and
// replaces the complete array ``pairsOfFlags``.
pairsOfFlags = newPairs;
}

struct StructType {
    uint[] contents;
    uint moreInfo;
}
StructType s;

function f(uint[] memory c) public {
    // stores a reference to ``s`` in ``g``
    StructType storage g = s;
    // also changes ``s.moreInfo``.
    g.moreInfo = 2;
    // assigns a copy because ``g.contents``
    // is not a local variable, but a member of
    // a local variable.
    g.contents = c;
}

function setFlagPair(uint index, bool flagA, bool flagB) public {
    // access to a non-existing index will throw an exception
    pairsOfFlags[index][0] = flagA;
    pairsOfFlags[index][1] = flagB;
}

function changeFlagArraySize(uint newSize) public {
    // using push and pop is the only way to change the
    // length of an array
    if (newSize < pairsOfFlags.length) {
        while (pairsOfFlags.length > newSize)
            pairsOfFlags.pop();
    } else if (newSize > pairsOfFlags.length) {
        while (pairsOfFlags.length < newSize)
            pairsOfFlags.push();
    }
}

function clear() public {
    // these clear the arrays completely
    delete pairsOfFlags;
    delete aLotOfIntegers;
    // identical effect here
    pairsOfFlags = new bool[2][](0);
}

bytes byteData;

function byteArrays(bytes memory data) public {
    // byte arrays ("bytes") are different as they are stored without padding,

```

(continues on next page)

(continued from previous page)

```
// but can be treated identical to "uint8[]"
byteData = data;
for (uint i = 0; i < 7; i++)
    byteData.push();
byteData[3] = 0x08;
delete byteData[2];
}

function addFlag(bool[2] memory flag) public returns (uint) {
    pairsOfFlags.push(flag);
    return pairsOfFlags.length;
}

function createMemoryArray(uint size) public pure returns (bytes memory) {
    // Dynamic memory arrays are created using `new`:
    uint[2][] memory arrayOfPairs = new uint[2][](size);

    // Inline arrays are always statically-sized and if you only
    // use literals, you have to provide at least one type.
    arrayOfPairs[0] = [uint(1), 2];

    // Create a dynamic byte array:
    bytes memory b = new bytes(200);
    for (uint i = 0; i < b.length; i++)
        b[i] = bytes1(uint8(i));
    return b;
}
}
```

Dangling References to Storage Array Elements

When working with storage arrays, you need to take care to avoid dangling references. A dangling reference is a reference that points to something that no longer exists or has been moved without updating the reference. A dangling reference can for example occur, if you store a reference to an array element in a local variable and then `.pop()` from the containing array:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[][] s;

    function f() public {
        // Stores a pointer to the last array element of s.
        uint[] storage ptr = s[s.length - 1];
        // Removes the last array element of s.
        s.pop();
        // Writes to the array element that is no longer within the array.
        ptr.push(0x42);
        // Adding a new element to ``s`` now will not add an empty array, but
        // will result in an array of length 1 with ``0x42`` as element.
    }
}
```

(continues on next page)

(continued from previous page)

```

        s.push();
        assert(s[s.length - 1][0] == 0x42);
    }
}

```

The write in `ptr.push(0x42)` will **not** revert, despite the fact that `ptr` no longer refers to a valid element of `s`. Since the compiler assumes that unused storage is always zeroed, a subsequent `s.push()` will not explicitly write zeroes to storage, so the last element of `s` after that `push()` will have length 1 and contain `0x42` as its first element.

Note that Solidity does not allow to declare references to value types in storage. These kinds of explicit dangling references are restricted to nested reference types. However, dangling references can also occur temporarily when using complex expressions in tuple assignments:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[] s;
    uint[] t;
    constructor() {
        // Push some initial values to the storage arrays.
        s.push(0x07);
        t.push(0x03);
    }

    function g() internal returns (uint[] storage) {
        s.pop();
        return t;
    }

    function f() public returns (uint[] memory) {
        // The following will first evaluate ``s.push()`` to a reference to a new element
        // at index 1. Afterwards, the call to ``g`` pops this new element, resulting in
        // the left-most tuple element to become a dangling reference. The assignment
        ↪ still
        // takes place and will write outside the data area of ``s``.
        (s.push(), g()[0]) = (0x42, 0x17);
        // A subsequent push to ``s`` will reveal the value written by the previous
        // statement, i.e. the last element of ``s`` at the end of this function will have
        // the value ``0x42``.
        s.push();
        return s;
    }
}

```

It is always safer to only assign to storage once per statement and to avoid complex expressions on the left-hand-side of an assignment.

You need to take particular care when dealing with references to elements of `bytes` arrays, since a `.push()` on a `bytes` array may switch *from short to long layout in storage*.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

```

(continues on next page)

(continued from previous page)

```
// This will report a warning
contract C {
    bytes x = "012345678901234567890123456789";

    function test() external returns(uint) {
        (x.push(), x.push()) = (0x01, 0x02);
        return x.length;
    }
}
```

Here, when the first `x.push()` is evaluated, `x` is still stored in short layout, thereby `x.push()` returns a reference to an element in the first storage slot of `x`. However, the second `x.push()` switches the bytes array to large layout. Now the element that `x.push()` referred to is in the data area of the array while the reference still points at its original location, which is now a part of the length field and the assignment will effectively garble the length of `x`. To be safe, only enlarge bytes arrays by at most one element during a single assignment and do not simultaneously index-access the array in the same statement.

While the above describes the behaviour of dangling storage references in the current version of the compiler, any code with dangling references should be considered to have *undefined behaviour*. In particular, this means that any future version of the compiler may change the behaviour of code that involves dangling references.

Be sure to avoid dangling references in your code!

Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a `uint256` type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional: `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

Note: As of now, array slices are only implemented for calldata arrays.

Array slices are useful to ABI-decode secondary data passed in function parameters:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;
contract Proxy {
    /// @dev Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address client_) {
        client = client_;
    }
}
```

(continues on next page)

(continued from previous page)

```

/// Forward call to "setOwner(address)" that is implemented by client
/// after doing basic validation on the address argument.
function forward(bytes calldata payload) external {
    bytes4 sig = bytes4(payload[:4]);
    // Due to truncating behaviour, bytes4(payload) performs identically.
    // bytes4 sig = bytes4(payload);
    if (sig == bytes4(keccak256("setOwner(address)"))) {
        address owner = abi.decode(payload[4:], (address));
        require(owner != address(0), "Address of owner cannot be zero.");
    }
    (bool status,) = client.delegatecall(payload);
    require(status, "Forwarded call failed.");
}
}

```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    // Structs can also be defined inside contracts, which makes them
    // visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address payable beneficiary, uint goal) public returns (uint
    ↵campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // We cannot use "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)"
        // because the right hand side creates a memory-struct "Campaign" that contains
    ↵a mapping.
}

```

(continues on next page)

(continued from previous page)

```

Campaign storage c = campaigns[campaignID];
c.beneficiary = beneficiary;
c.fundingGoal = goal;
}

function contribute(uint campaignID) public payable {
    Campaign storage c = campaigns[campaignID];
    // Creates a new temporary memory struct, initialised with the given values
    // and copies it over to storage.
    // Note that you can also use Funder(msg.sender, msg.value) to initialise.
    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
    c.amount += msg.value;
}

function checkGoalReached(uint campaignID) public returns (bool reached) {
    Campaign storage c = campaigns[campaignID];
    if (c.amount < c.fundingGoal)
        return false;
    uint amount = c.amount;
    c.amount = 0;
    c.beneficiary.transfer(amount);
    return true;
}
}

```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable with data location `storage`. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

Note: Until Solidity 0.7.0, memory-structs containing members of storage-only types (e.g. mappings) were allowed and assignments like `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` in the example above would work and just silently skip those members.

3.6.3 Mapping Types

Mapping types use the syntax `mapping(KeyType => ValueType)` and variables of mapping type are declared using the syntax `mapping(KeyType => ValueType) VariableName`. The `KeyType` can be any built-in value type, `bytes`, `string`, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. `ValueType` can be any type, including mappings, arrays and structs.

You can think of mappings as [hash tables](#), which are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's *default value*. The similarity ends there, the key data is not stored in a mapping, only its keccak256 hash is used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information regarding the assigned keys (see [Clearing Mappings](#)).

Mappings can only have a data location of `storage` and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.

You can mark state variables of mapping type as `public` and Solidity creates a [getter](#) for you. The `KeyType` becomes a parameter for the getter. If `ValueType` is a value type or a struct, the getter returns `ValueType`. If `ValueType` is an array or a mapping, the getter has one parameter for each `KeyType`, recursively.

In the example below, the `MappingExample` contract defines a public `balances` mapping, with the key type an address, and a value type a `uint`, mapping an Ethereum address to an unsigned integer value. As `uint` is a value type, the getter returns a value that matches the type, which you can see in the `MappingUser` contract that returns the value at the specified address.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

The example below is a simplified version of an [ERC20 token](#). `_allowances` is an example of a mapping type inside another mapping type. The example below uses `_allowances` to record the amount someone else is allowed to withdraw from your account.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
```

(continues on next page)

(continued from previous page)

```

mapping (address => mapping (address => uint256)) private _allowances;

event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed owner, address indexed spender, uint256 value);

function allowance(address owner, address spender) public view returns (uint256) {
    return _allowances[owner][spender];
}

function transferFrom(address sender, address recipient, uint256 amount) public
returns (bool) {
    require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high
enough.");
    _allowances[sender][msg.sender] -= amount;
    _transfer(sender, recipient, amount);
    return true;
}

function approve(address spender, uint256 amount) public returns (bool) {
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

function _transfer(address sender, address recipient, uint256 amount) internal {
    require(sender != address(0), "ERC20: transfer from the zero address");
    require(recipient != address(0), "ERC20: transfer to the zero address");
    require(_balances[sender] >= amount, "ERC20: Not enough funds");

    _balances[sender] -= amount;
    _balances[recipient] += amount;
    emit Transfer(sender, recipient, amount);
}
}

```

Iterable Mappings

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that. For example, the code below implements an `IterableMapping` library that the User contract then adds data to, and the `sum` function iterates over to sum all the values.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
}

```

(continues on next page)

(continued from previous page)

```

KeyFlag[] keys;
uint size;
}

type Iterator is uint;

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool)
→replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size--;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterateStart(itmap storage self) internal view returns (Iterator) {
        return iteratorSkipDeleted(self, 0);
    }

    function iterateValid(itmap storage self, Iterator iterator) internal view returns_
→(bool) {
        return Iterator.unwrap(iterator) < self.keys.length;
    }

    function iterateNext(itmap storage self, Iterator iterator) internal view returns_
→(Iterator) {
        return iteratorSkipDeleted(self, Iterator.unwrap(iterator) + 1);
    }

    function iterateGet(itmap storage self, Iterator iterator) internal view returns_
→(uint key, uint value) {
}

```

(continues on next page)

(continued from previous page)

```

uint keyIndex = Iterator.unwrap(iterator);
key = self.keys[keyIndex].key;
value = self.data[key].value;
}

function iteratorSkipDeleted(itmap storage self, uint keyIndex) private view returns_
(Iterator) {
    while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
        keyIndex++;
    return Iterator.wrap(keyIndex);
}

// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            Iterator i = data.iterateStart();
            data.iterateValid(i);
            i = data.iterateNext(i)
        ) {
            (, uint value) = data.iterateGet(i);
            s += value;
        }
    }
}

```

3.6.4 Operators

Arithmetic and bit operators can be applied even if the two operands do not have the same type. For example, you can compute `y = x + z`, where `x` is a `uint8` and `z` has the type `int32`. In these cases, the following mechanism will be used to determine the type in which the operation is computed (this is important in case of overflow) and the type of the operator's result:

1. If the type of the right operand can be implicitly converted to the type of the left operand, use the type of the left operand,
2. if the type of the left operand can be implicitly converted to the type of the right operand, use the type of the right operand,
3. otherwise, the operation is not allowed.

In case one of the operands is a *literal number* it is first converted to its “mobile type”, which is the smallest type that can hold the value (unsigned types of the same bit-width are considered “smaller” than the signed types). If both are literal numbers, the operation is computed with arbitrary precision.

The operator's result type is the same as the type the operation is performed in, except for comparison operators where the result is always `bool`.

The operators `**` (exponentiation), `<<` and `>>` use the type of the left operand for the operation and the result.

Ternary Operator

The ternary operator is used in expressions of the form `<expression> ? <trueExpression> : <falseExpression>`. It evaluates one of the latter two given expressions depending upon the result of the evaluation of the main `<expression>`. If `<expression>` evaluates to `true`, then `<trueExpression>` will be evaluated, otherwise `<falseExpression>` is evaluated.

The result of the ternary operator does not have a rational number type, even if all of its operands are rational number literals. The result type is determined from the types of the two operands in the same way as above, converting to their mobile type first if required.

As a consequence, `255 + (true ? 1 : 0)` will revert due to arithmetic overflow. The reason is that `(true ? 1 : 0)` is of `uint8` type, which forces the addition to be performed in `uint8` as well, and 256 exceeds the range allowed for this type.

Another consequence is that an expression like `1.5 + 1.5` is valid but `1.5 + (true ? 1.5 : 2.5)` is not. This is because the former is a rational expression evaluated in unlimited precision and only its final value matters. The latter involves a conversion of a fractional rational number to an integer, which is currently disallowed.

Compound and Increment/Decrement Operators

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as short-hands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=` and `>>=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` / `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a *mapping* is probably a better choice.

For structs, it assigns a struct with all members reset. In other words, the value of `a` after `delete a` is the same as if `a` would be declared without assignment, with the following caveat:

`delete` has no effect on mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted: If `a` is a mapping, then `delete a[x]` will delete the value stored at `x`.

It is important to note that `delete a` really behaves like an assignment to `a`, i.e. it stores a new object in `a`. This distinction is visible when `a` is reference variable: It will only reset `a` itself, not the value it referred to previously.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a
        ↪complex object, also
            // y is affected which is an alias to the storage object
            // On the other hand: "delete y" is not valid, as assignments to local variables
            // referencing storage objects can only be made from existing storage objects.
        assert(y.length == 0);
    }
}
```

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	<code>++, --</code>
	New expression	<code>new <typename></code>
	Array subscripting	<code><array>[<index>]</code>
	Member access	<code><object>. <member></code>
	Function-like call	<code><func>(<args...>)</code>
	Parentheses	<code>(<statement>)</code>
2	Prefix increment and decrement	<code>++, --</code>
	Unary minus	<code>-</code>
	Unary operations	<code>delete</code>
	Logical NOT	<code>!</code>
	Bitwise NOT	<code>~</code>
3	Exponentiation	<code>**</code>
4	Multiplication, division and modulo	<code>*, /, %</code>
5	Addition and subtraction	<code>+, -</code>
6	Bitwise shift operators	<code><<, >></code>
7	Bitwise AND	<code>&</code>
8	Bitwise XOR	<code>^</code>
9	Bitwise OR	<code> </code>
10	Inequality operators	<code><, >, <=, >=</code>
11	Equality operators	<code>==, !=</code>
12	Logical AND	<code>&&</code>
13	Logical OR	<code> </code>
14	Ternary operator	<code><conditional> ? <if-true> : <if-false></code>
	Assignment operators	<code>=, =, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=</code>
15	Comma operator	<code>,</code>

3.6.5 Conversions between Elementary Types

Implicit Conversions

An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

For example, `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256`, because `uint256` cannot hold values such as `-1`.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). This means that operations are always performed in the type of one of the operands.

For more details about which implicit conversions are possible, please consult the sections about the types themselves.

In the example below, `y` and `z`, the operands of the addition, do not have the same type, but `uint8` can be implicitly converted to `uint16` and not vice-versa. Because of that, `y` is converted to the type of `z` before the addition is performed in the `uint16` type. The resulting type of the expression `y + z` is `uint16`. Because it is assigned to a variable of type `uint32` another implicit conversion is performed after the addition.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

Explicit Conversions

If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler, so be sure to test that the result is what you want and expect!

Take the following example that converts a negative `int` to a `uint`:

```
int y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If an integer is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end). The result of the conversion will compare equal to the original integer:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Fixed-size bytes types behave differently during conversions. They can be thought of as sequences of individual bytes and converting to a smaller type will cut off the sequence:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b will be 0x12
```

If a fixed-size bytes type is explicitly converted to a larger type, it is padded on the right. Accessing the byte at a fixed index will result in the same value before and after the conversion (if the index is still in range):

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Since integers and fixed-size byte arrays behave differently when truncating or padding, explicit conversions between integers and fixed-size byte arrays are only allowed, if both have the same size. If you want to convert between integers and fixed-size byte arrays of different size, you have to use intermediate conversions that make the desired truncation and padding rules explicit:

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

`bytes` arrays and `bytes` calldata slices can be converted explicitly to fixed bytes types (`bytes1`/.../`bytes32`). In case the array is longer than the target fixed bytes type, truncation at the end will happen. If the array is shorter than the target type, it will be padded with zeros at the end.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16, bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m); // if length of m is greater than 16, truncation will
        ↪happen
        b = bytes16(s); // padded on the right, so result is "abcdefgh\0\0\0\0\0\0\0\0\0"
        bytes3 b1 = bytes3(s); // truncated, b1 equals to "abc"
        b = bytes16(c[:8]); // also padded with zeros
        return (b, b1);
    }
}
```

3.6.6 Conversions between Literals and Elementary Types

Integer Types

Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation:

```
uint8 a = 12; // fine
uint32 b = 1234; // fine
uint16 c = 0x123456; // fails, since it would have to truncate to 0x3456
```

Note: Prior to version 0.8.0, any decimal or hexadecimal number literals could be explicitly converted to an integer type. From 0.8.0, such explicit conversions are as strict as implicit conversions, i.e., they are only allowed if the literal fits in the resulting range.

Fixed-Size Byte Arrays

Decimal number literals cannot be implicitly converted to fixed-size byte arrays. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the `bytes` type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type:

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the `bytes` type:

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

Addresses

As described in [Address Literals](#), hex literals of the correct size that pass the checksum test are of address type. No other literals can be implicitly converted to the address type.

Explicit conversions from bytes20 or any integer type to address result in address payable.

An address a can be converted to address payable via payable(a).

3.7 Units and Globally Available Variables

3.7.1 Ether Units

A literal number can take a suffix of wei, gwei or ether to specify a subdenomination of Ether, where Ether numbers without a postfix are assumed to be Wei.

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

The only effect of the subdenomination suffix is a multiplication by a power of ten.

Note: The denominations finney and szabo have been removed in version 0.7.0.

3.7.2 Time Units

Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks == 7 days

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of [leap seconds](#). Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

Note: The suffix years has been removed in version 0.5.0 due to the reasons above.

These suffixes cannot be applied to variables. For example, if you want to interpret a function parameter in days, you can in the following way:

```
function f(uint start, uint daysAfter) public {
    if (block.timestamp >= start + daysAfter * 1 days) {
        // ...
    }
}
```

3.7.3 Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

Block and Transaction Properties

- `blockhash(uint blockNumber)` returns (bytes32): hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `block.basefee (uint)`: current block's base fee ([EIP-3198](#) and [EIP-1559](#))
- `block.chainid (uint)`: current chain id
- `block.coinbase (address payable)`: current block miner's address
- `block.difficulty (uint)`: current block difficulty
- `block.gaslimit (uint)`: current block gaslimit
- `block.number (uint)`: current block number
- `block.timestamp (uint)`: current block timestamp as seconds since unix epoch
- `gasleft()` returns (uint256): remaining gas
- `msg.data (bytes calldata)`: complete calldata
- `msg.sender (address)`: sender of the message (current call)
- `msg.sig (bytes4)`: first four bytes of the calldata (i.e. function identifier)
- `msg.value (uint)`: number of wei sent with the message
- `tx.gasprice (uint)`: gas price of the transaction
- `tx.origin (address)`: sender of the transaction (full call chain)

Note: The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

Note: When contracts are evaluated off-chain rather than in context of a transaction included in a block, you should not assume that `block.*` and `tx.*` refer to values from any specific block or transaction. These values are provided by the EVM implementation that executes the contract and can be arbitrary.

Note: Do not rely on `block.timestamp` or `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note: The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note: The function `blockhash` was previously known as `block.blockhash`, which was deprecated in version 0.4.22 and removed in version 0.5.0.

Note: The function `gasleft` was previously known as `msg.gas`, which was deprecated in version 0.4.21 and removed in version 0.5.0.

Note: In version 0.7.0, the alias `now` (for `block.timestamp`) was removed.

ABI Encoding and Decoding Functions

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-decodes the given data, while the types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...)` returns `(bytes memory)`: ABI-encodes the given arguments
- `abi.encodePacked(...)` returns `(bytes memory)`: Performs *packed encoding* of the given arguments. Note that packed encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...)` returns `(bytes memory)`: ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...)` returns `(bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)))), ...)`
- `abi.encodeCall(function functionPointer, ...)` returns `(bytes memory)`: ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, (...))`

Note: These encoding functions can be used to craft data for external function calls without actually calling an external function. Furthermore, `keccak256(abi.encodePacked(a, b))` is a way to compute the hash of structured data (although be aware that it is possible to craft a “hash collision” using different function parameter types).

See the documentation about the [ABI](#) and the [tightly packed encoding](#) for details about the encoding.

Members of bytes

- `bytes.concat(...)` returns `(bytes memory)`: *Concatenates variable number of bytes and bytes1, ..., bytes32 arguments to one byte array*

Members of string

- `string.concat(...)` returns `(string memory)`: *Concatenates variable number of string arguments to one string array*

Error Handling

See the dedicated section on [assert and require](#) for more details on error handling and when to use which function.

`assert(bool condition)`

causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.

`require(bool condition)`

reverts if the condition is not met - to be used for errors in inputs or external components.

`require(bool condition, string memory message)`

reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

`revert()`

abort execution and revert state changes

`revert(string memory reason)`

abort execution and revert state changes, providing an explanatory string

Mathematical and Cryptographic Functions

`addmod(uint x, uint y, uint k) returns (uint)`

compute $(x + y) \% k$ where the addition is performed with arbitrary precision and does not wrap around at 2^{**256} . Assert that $k \neq 0$ starting from version 0.5.0.

`mulmod(uint x, uint y, uint k) returns (uint)`

compute $(x * y) \% k$ where the multiplication is performed with arbitrary precision and does not wrap around at 2^{**256} . Assert that $k \neq 0$ starting from version 0.5.0.

`keccak256(bytes memory) returns (bytes32)`

compute the Keccak-256 hash of the input

Note: There used to be an alias for keccak256 called sha3, which was removed in version 0.5.0.

`sha256(bytes memory) returns (bytes32)`

compute the SHA-256 hash of the input

`ripemd160(bytes memory) returns (bytes20)`

compute RIPEMD-160 hash of the input

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`

recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature:

- r = first 32 bytes of signature

- s = second 32 bytes of signature
- v = final 1 byte of signature

`ecrecover` returns an address, and not an address `payable`. See [address payable](#) for conversion, in case you need to transfer funds to the recovered address.

For further details, read [example usage](#).

Warning: If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for `_transaction_` signatures (see [EIP-2](#)), but the `ecrecover` function remained unchanged.

This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin have a [ECDSA helper library](#) that you can use as a wrapper for `ecrecover` without this issue.

Note: When running `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*, you might encounter Out-of-Gas. This is because these functions are implemented as “precompiled contracts” and only really exist after they receive the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution might run into an Out-of-Gas error. A workaround for this problem is to first send Wei (1 for example) to each of the contracts before you use them in your actual contracts. This is not an issue on the main or test net.

Members of Address Types

<address>.balance (uint256)

balance of the [Address](#) in Wei

<address>.code (bytes memory)

code at the [Address](#) (can be empty)

<address>.codehash (bytes32)

the codehash of the [Address](#)

<address payable>.transfer(uint256 amount)

send given amount of Wei to [Address](#), reverts on failure, forwards 2300 gas stipend, not adjustable

<address payable>.send(uint256 amount) returns (bool)

send given amount of Wei to [Address](#), returns false on failure, forwards 2300 gas stipend, not adjustable

<address>.call(bytes memory) returns (bool, bytes memory)

issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

<address>.delegatecall(bytes memory) returns (bool, bytes memory)

issue low-level DELEGATECALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

<address>.staticcall(bytes memory) returns (bool, bytes memory)

issue low-level STATICCALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

For more information, see the section on [Address](#).

Warning: You should avoid using `.call()` whenever possible when executing another contract function as it bypasses type checking, function existence check, and argument packing.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: Use a pattern where the recipient withdraws the money.

Warning: Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity includes an extra check using the `extcodesize` opcode when performing external calls. This ensures that the contract that is about to be called either actually exists (it contains code) or an exception is raised.

The low-level calls which operate on addresses rather than contract instances (i.e. `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` and `.transfer()`) **do not** include this check, which makes them cheaper in terms of gas but also less safe.

Note: Prior to version 0.5.0, Solidity allowed address members to be accessed by a contract instance, for example `this.balance`. This is now forbidden and an explicit conversion to address must be done: `address(this).balance`.

Note: If state variables are accessed via a low-level `delegatecall`, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

Note: Prior to version 0.5.0, `.call`, `.delegatecall` and `.staticcall` only returned the success condition and not the return data.

Note: Prior to version 0.5.0, there was a member called `callcode` with similar but slightly different semantics than `delegatecall`.

Contract Related

`this` (current contract's type)

the current contract, explicitly convertible to `Address`

`selfdestruct(address payable recipient)`

Destroy the current contract, sending its funds to the given `Address` and end execution. Note that `selfdestruct` has some peculiarities inherited from the EVM:

- the receiving contract's receive function is not executed.
- the contract is only really destroyed at the end of the transaction and `revert`s might "undo" the destruction.

Furthermore, all functions of the current contract are callable directly including the current function.

Note: Prior to version 0.5.0, there was a function called `suicide` with the same semantics as `selfdestruct`.

Type Information

The expression `type(X)` can be used to retrieve information about the type X. Currently, there is limited support for this feature (X can be either a contract or an integer type) but it might be expanded in the future.

The following properties are available for a contract type C:

`type(C).name`

The name of the contract.

`type(C).creationCode`

Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `create2` opcode. This property can **not** be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

`type(C).runtimeCode`

Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of C. If C has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with `.creationCode` also apply for this property.

In addition to the properties above, the following properties are available for an interface type I:

`type(I).interfaceId`:

A bytes4 value containing the [EIP-165](#) interface identifier of the given interface I. This identifier is defined as the XOR of all function selectors defined within the interface itself - excluding all inherited functions.

The following properties are available for an integer type T:

`type(T).min`

The smallest value representable by type T.

`type(T).max`

The largest value representable by type T.

3.8 Expressions and Control Structures

3.8.1 Control Structures

Most of the control structures known from curly-braces languages are available in Solidity:

There is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from C or JavaScript.

Solidity also supports exception handling in the form of `try/catch`-statements, but only for [external function calls](#) and contract creation calls. Errors can be created using the [revert statement](#).

Parentheses can *not* be omitted for conditionals, but curly braces can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

3.8.2 Function Calls

Internal Function Calls

Functions of the current contract can be called directly (“internally”), also recursively, as seen in this nonsensical example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

// This will report a warning
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract instance can be called internally.

You should still avoid excessive recursion, as every internal function call uses up at least one stack slot and there are only 1024 slots available.

External Function Calls

Functions can also be called using the `this.g(8);` and `c.g(2);` notation, where `c` is a contract instance and `g` is a function belonging to `c`. Calling the function `g` via either way results in it being called “externally”, using a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

Note: A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

When calling functions of other contracts, you can specify the amount of Wei or gas sent with the call with the special options `{value: 10, gas: 10000}`. Note that it is discouraged to specify gas values explicitly, since the gas costs of opcodes can change in the future. Any Wei you send to the contract is added to the total balance of that contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

You need to use the modifier `payable` with the `info` function because otherwise, the `value` option would not be available.

Warning: Be careful that `feed.info{value: 10, gas: 800}` only locally sets the `value` and amount of `gas` sent with the function call, and the parentheses at the end perform the actual call. So `feed.info{value: 10, gas: 800}` does not call the function and the `value` and `gas` settings are lost, only `feed.info{value: 10, gas: 800}()` performs the function call.

Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity uses the `extcodesize` opcode to check that the contract that is about to be called actually exists (it contains code) and causes an exception if it does not. This check is skipped if the return data will be decoded after the call and thus the ABI decoder will catch the case of a non-existing contract.

Note that this check is not performed in case of *low-level calls* which operate on addresses rather than contract instances.

Note: Be careful when using high-level calls to *precompiled contracts*, since the compiler considers them non-existing according to the above logic even though they execute code and can return data.

Function calls also cause exceptions if the called contract itself throws an exception or goes out of gas.

Warning: Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Note: Before Solidity 0.6.2, the recommended way to specify the `value` and `gas` was to use `f.value(x).gas(g)()`. This was deprecated in Solidity 0.6.2 and is no longer possible since Solidity 0.7.0.

Named Calls and Anonymous Function Parameters

Function call arguments can be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}

```

Omitted Function Parameter Names

The names of unused parameters (especially return parameters) can be omitted. Those parameters will still be present on the stack, but they are inaccessible.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}

```

3.8.3 Creating Contracts via new

A contract can create other contracts using the `new` keyword. The full code of the contract being created has to be known when the creating contract is compiled so recursive creation-dependencies are not possible.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

```

As seen in the example, it is possible to send Ether while creating an instance of D using the `value` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Salted contract creations / `create2`

When creating a contract, the address of the contract is computed from the address of the creating contract and a counter that is increased with each contract creation.

If you specify the option `salt` (a `bytes32` value), then contract creation will use a different mechanism to come up with the address of the new contract:

It will compute the address from the address of the creating contract, the given salt value, the (creation) bytecode of the created contract and the constructor arguments.

In particular, the counter (“nonce”) is not used. This allows for more flexibility in creating contracts: You are able to derive the address of the new contract before it is created. Furthermore, you can rely on this address also in case the creating contracts creates other contracts in the meantime.

The main use-case here is contracts that act as judges for off-chain interactions, which only need to be created if there is a dispute.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // This complicated expression just tells you how the address
        // can be pre-computed. It is just there for illustration.
        // You actually only need `new D{salt: salt}(arg)`.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                abi.encode(arg)
            )));
        ))));
        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Warning: There are some peculiarities in relation to salted creation. A contract can be re-created at the same address after having been destroyed. Yet, it is possible for that newly created contract to have a different deployed bytecode even though the creation bytecode has been the same (which is a requirement because otherwise the address would change). This is due to the fact that the constructor can query external state that might have changed between the two creations and incorporate that into the deployed bytecode before it is stored.

3.8.4 Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

3.8.5 Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose number is a constant at compile-time. Those tuples can be used to return multiple values at the same time. These can then either be assigned to newly declared variables or to pre-existing variables (or LValues in general).

Tuples are not proper types in Solidity, they can only be used to form syntactic groupings of expressions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid: `(x, uint y) = (1, 2);`

Note: Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

Warning: Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behaviour.

Complications for Arrays and Structs

The semantics of assignments are more complicated for non-value types like arrays and structs, including `bytes` and `string`, see [Data location and assignment behaviour](#) for details.

In the example below the call to `g(x)` has no effect on `x` because it creates an independent copy of the storage value in memory. However, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The “default values” of variables are the typical “zero-state” of whatever the type is. For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or string. For the `enum` type, the default value is its first member.

Scoping in Solidity follows the widespread scoping rules of C99 (and many other languages): Variables are visible from the point right after their declaration until the end of the smallest `{ }`-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.

Variables that are parameter-like (function parameters, modifier parameters, catch parameters, ...) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.

Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

As a consequence, the following examples will compile without warnings, since the two variables have the same name but disjoint scopes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

As a special example of the C99 scoping rules, note that in the following, the first assignment to `x` will actually assign the outer and not the inner variable. In any case, you will get a warning about the outer variable being shadowed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

Warning: Before version 0.5.0 Solidity followed the same scoping rules as JavaScript, that is, a variable declared anywhere within a function would be in scope for the entire function, regardless where it was declared. The following example shows a code snippet that used to compile but leads to an error starting from version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 Checked or Unchecked Arithmetic

An overflow or underflow is the situation where the resulting value of an arithmetic operation, when executed on an unrestricted integer, falls outside the range of the result type.

Prior to Solidity 0.8.0, arithmetic operations would always wrap in case of under- or overflow leading to widespread use of libraries that introduce additional checks.

Since Solidity 0.8.0, all arithmetic operations revert on over- and underflow by default, thus making the use of these libraries unnecessary.

To obtain the previous behaviour, an unchecked block can be used:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // This subtraction will wrap on underflow.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // This subtraction will revert on underflow.
        return a - b;
    }
}
```

The call to `f(2, 3)` will return `2**256-1`, while `g(2, 3)` will cause a failing assertion.

The unchecked block can be used everywhere inside a block, but not as a replacement for a block. It also cannot be nested.

The setting only affects the statements that are syntactically inside the block. Functions called from within an unchecked block do not inherit the property.

Note: To avoid ambiguity, you cannot use `_;` inside an unchecked block.

The following operators will cause a failing assertion on overflow or underflow and will wrap without an error if used inside an unchecked block:

`++, --, +, binary -, unary -, *, /, %, **`
`+=, -=, *=, /=, %=`

Warning: It is not possible to disable the check for division by zero or modulo by zero using the unchecked block.

Note: Bitwise operators do not perform overflow or underflow checks. This is particularly visible when using bitwise shifts (`<<, >>, <<=, >>=`) in place of integer division and multiplication by a power of 2. For example `type(uint256).max << 3` does not revert even though `type(uint256).max * 8` would.

Note: The second statement in `int x = type(int).min; -x;` will result in an overflow because the negative range can hold one more value than the positive range.

Explicit type conversions will always truncate and never cause a failing assertion with the exception of a conversion from an integer to an enum type.

3.8.8 Error handling: Assert, Require, Revert and Exceptions

Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller.

When exceptions happen in a sub-call, they “bubble up” (i.e., exceptions are rethrown) automatically unless they are caught in a `try/catch` statement. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall`: they return `false` as their first return value in case of an exception instead of “bubbling up”.

Warning: The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

Exceptions can contain error data that is passed back to the caller in the form of *error instances*. The built-in errors `Error(string)` and `Panic(uint256)` are used by special functions, as explained below. `Error` is used for “regular” error conditions while `Panic` is used for errors that should not be present in bug-free code.

Panic via assert and Error via require

The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met.

The `assert` function creates an error of type `Panic(uint256)`. The same error is created by the compiler in certain situations as listed below.

Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.

A Panic exception is generated in the following situations. The error code supplied with the error data indicates the kind of panic.

1. 0x00: Used for generic compiler inserted panics.
2. 0x01: If you call `assert` with an argument that evaluates to false.
3. 0x11: If an arithmetic operation results in underflow or overflow outside of an `unchecked { ... }` block.
4. 0x12: If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
5. 0x21: If you convert a value that is too big or negative into an enum type.
6. 0x22: If you access a storage byte array that is incorrectly encoded.
7. 0x31: If you call `.pop()` on an empty array.
8. 0x32: If you access an array, `bytesN` or an array slice at an out-of-bounds or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
9. 0x41: If you allocate too much memory or create an array that is too large.
10. 0x51: If you call a zero-initialized variable of internal function type.

The `require` function either creates an error without any data or an error of type `Error(string)`. It should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

Note: It is currently not possible to use custom errors in combination with `require`. Please use `if (!condition) revert CustomError();` instead.

An `Error(string)` exception (or an exception without data) is generated by the compiler in the following situations:

1. Calling `require(x)` where `x` evaluates to `false`.
2. If you use `revert()` or `revert("description")`.
3. If you perform an external function call targeting a contract that contains no code.
4. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
5. If your contract receives Ether via a public getter function.

For the following cases, the error data from the external call (if provided) is forwarded. This means that it can either cause an `Error` or a `Panic` (or whatever else was given):

1. If a `.transfer()` fails.
2. If you call a function via a message call but it does not finish properly (i.e., it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall`, `callcode` or `staticcall` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
3. If you create a contract using the `new` keyword but the contract creation *does not finish properly*.

You can optionally provide a message string for `require`, but not for `assert`.

Note: If you do not provide a string argument to `require`, it will revert with empty error data, not even including the error selector.

The following example shows how you can use `require` to check conditions on inputs and `assert` for internal error checking.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

Internally, Solidity performs a revert operation (instruction `0xfd`). This causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to keep the atomicity of transactions, the safest action is to revert all changes and make the whole transaction (or at least call) without effect.

In both cases, the caller can react on such failures using `try/catch`, but the changes in the callee will always be reverted.

Note: Panic exceptions used to use the `invalid` opcode before Solidity 0.8.0, which consumed all gas available to the call. Exceptions that use `require` used to consume all gas until before the Metropolis release.

revert

A direct revert can be triggered using the `revert` statement and the `revert` function.

The `revert` statement takes a custom error as direct argument without parentheses:

```
revert CustomError(arg1, arg2);
```

For backwards-compatibility reasons, there is also the `revert()` function, which uses parentheses and accepts a string:

```
revert(); revert("description");
```

The error data will be passed back to the caller and can be caught there. Using `revert()` causes a revert without any error data while `revert("description")` will create an `Error(string)` error.

Using a custom error instance will usually be much cheaper than a string description, because you can use the name of the error to describe it, which is encoded in only four bytes. A longer description can be supplied via NatSpec which does not incur any costs.

The following example shows how to use an error string and a custom error instance together with `revert` and the equivalent `require`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();
        payable(msg.sender).transfer(address(this).balance);
    }
}
```

The two ways `if (!condition) revert(...);` and `require(condition, ...)`; are equivalent as long as the arguments to `revert` and `require` do not have side-effects, for example if they are just strings.

Note: The `require` function is evaluated just as any other function. This means that all arguments are evaluated before the function itself is executed. In particular, in `require(condition, f())` the function `f` is executed even if `condition` is true.

The provided string is *abi-encoded* as if it were a call to a function `Error(string)`. In the above example, `revert("Not enough Ether provided.");` returns the following hexadecimal as error return data:

The provided message can be retrieved by the caller using `try/catch` as shown below.

Note: There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

try/catch

A failure in an external call can be caught using a try/catch statement, as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic,
            // i.e. a serious error like division by zero
            // or overflow. The error code can be used
            // to determine the kind of error.
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        errorCount++;
        return (0, false);
    } catch (bytes memory /*lowLevelData*/) {
        // This is executed in case revert() was used.
        errorCount++;
        return (0, false);
    }
}
}

```

The `try` keyword has to be followed by an expression representing an external function call or a contract creation (`new ContractName()`). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The `returns` part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the `catch` blocks.

Solidity supports different kinds of catch blocks depending on the type of error:

- `catch Error(string memory reason) { ... }`: This catch clause is executed if the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception).
- `catch Panic(uint errorCode) { ... }`: If the error was caused by a panic, i.e. by a failing `assert`, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
- `catch (bytes memory lowLevelData) { ... }`: This clause is executed if the error signature does not match any other clause, if there was an error while decoding the error message, or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.
- `catch { ... }`: If you are not interested in the error data, you can just use `catch { ... }` (even as the only catch clause) instead of the previous clause.

It is planned to support other types of error data in the future. The strings `Error` and `Panic` are currently parsed as is and are not treated as identifiers.

In order to catch all error cases, you have to have at least the clause `catch { ... }` or the clause `catch (bytes memory lowLevelData) { ... }`.

The variables declared in the `returns` and the `catch` clause are only in scope in the block that follows.

Note: If an error happens during the decoding of the return data inside a try/catch-statement, this causes an exception in the currently executing contract and because of that, it is not caught in the catch clause. If there is an error during decoding of `catch Error(string memory reason)` and there is a low-level catch clause, this error is caught there.

Note: If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

Note: The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded

it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: The caller always retains at least 1/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left.

3.9 Contracts

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables, and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible. A contract and its functions need to be called for anything to happen. There is no “cron” concept in Ethereum to call a function at a particular event automatically.

3.9.1 Creating Contracts

Contracts can be created “from outside” via Ethereum transactions or from within Solidity contracts.

IDEs, such as [Remix](#), make the creation process seamless using UI elements.

One way to create contracts programmatically on Ethereum is via the JavaScript API [web3.js](#). It has a function called `web3.eth.Contract` to facilitate contract creation.

When a contract is created, its `constructor` (a function declared with the `constructor` keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed *ABI encoded* after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 name_) {
        // State variables are accessed via their name
        // and not via e.g. `this.owner`. Functions can
        // be accessed directly or through `this.f`,
        // but the latter provides an external view
```

(continues on next page)

(continued from previous page)

```

// to the function. Especially in the constructor,
// you should not access functions externally,
// because the function does not exist yet.
// See the next section for details.
owner = msg.sender;

// We perform an explicit type conversion from `address`
// to `TokenCreator` and assume that the type of
// the calling contract is `TokenCreator`, there is
// no real way to verify that.
// This does not create a new contract.
creator = TokenCreator(msg.sender);
name = name_;

}

function changeName(bytes32 newName) public {
    // Only the creator can alter the name.
    // We compare the contract based on its
    // address which can be retrieved by
    // explicit conversion to address.
    if (msg.sender == address(creator))
        name = newName;
}

function transfer(address newOwner) public {
    // Only the current owner can transfer the token.
    if (msg.sender != owner) return;

    // We ask the creator contract if the transfer
    // should proceed by using a function of the
    // `TokenCreator` contract defined below. If
    // the call fails (e.g. due to out-of-gas),
    // the execution also fails here.
    if (creator.isTokenTransferOK(owner, newOwner))
        owner = newOwner;
}

contract TokenCreator {
    function createToken(bytes32 name)
        public
        returns (OwnedToken tokenAddress)
    {
        // Create a new `Token` contract and return its address.
        // From the JavaScript side, the return type
        // of this function is `address`, as this is
        // the closest type available in the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
}

```

(continues on next page)

(continued from previous page)

```

// Again, the external type of `tokenAddress` is
// simply `address`.
tokenAddress.changeName(name);
}

// Perform checks to determine if transferring a token to the
// `OwnedToken` contract should proceed
function isTokenTransferOK(address currentOwner, address newOwner)
public
pure
returns (bool ok)
{
    // Check an arbitrary condition to see if transfer should proceed
    return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
}
}

```

3.9.2 Visibility and Getters

State Variable Visibility

public

Public state variables differ from internal ones only in that the compiler automatically generates *getter functions* for them, which allows other contracts to read their values. When used within the same contract, the external access (e.g. `this.x`) invokes the getter while internal access (e.g. `x`) gets the variable value directly from storage. Setter functions are not generated so other contracts cannot directly modify their values.

internal

Internal state variables can only be accessed from within the contract they are defined in and in derived contracts. They cannot be accessed externally. This is the default visibility level for state variables.

private

Private state variables are like internal ones but they are not visible in derived contracts.

Warning: Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

Function Visibility

Solidity knows two kinds of function calls: external ones that do create an actual EVM message call and internal ones that do not. Furthermore, internal functions can be made inaccessible to derived contracts. This gives rise to four types of visibility for functions.

external

External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).

public

Public functions are part of the contract interface and can be either called internally or via message calls.

internal

Internal functions can only be accessed from within the current contract or contracts deriving from it. They

cannot be accessed externally. Since they are not exposed to the outside through the contract's ABI, they can take parameters of internal types like mappings or storage references.

private

Private functions are like internal ones but they are not visible in derived contracts.

Warning: Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, D, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract E is derived from C and, thus, can call `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

// This will not compile
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // error: member `f` is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member `compute` is not visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // access to internal member (from derived to parent)
    }
}
```

(continues on next page)

(continued from previous page)

}

Getter Functions

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. State variables can be initialized when they are declared.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it evaluates to a state variable. If it is accessed externally (i.e. with `this.`), it evaluates to a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal access
        return this.data(); // external access
    }
}
```

If you have a `public` state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array. You can use arguments to specify which individual element to return, for example `myArray(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // public state variable
    uint[] public myArray;

    // Getter function generated by the compiler
    /*
    function myArray(uint i) public view returns (uint) {

```

(continues on next page)

(continued from previous page)

```

        return myArray[i];
    }
}

// function that returns entire array
function getArray() public view returns (uint[] memory) {
    return myArray;
}
}

```

Now you can use `getArray()` to retrieve the entire array, instead of `myArray(i)`, which returns a single element per call.

The next example is more complex:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}

```

It generates a function of the following form. The mapping and arrays (with the exception of byte arrays) in the struct are omitted because there is no good way to select individual struct members or provide a key for the mapping:

```

function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}

```

3.9.3 Function Modifiers

Modifiers can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function.

Modifiers are inheritable properties of contracts and may be overridden by derived contracts, but only if they are marked `virtual`. For details, please see [Modifier Overriding](#).

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

```

(continues on next page)

(continued from previous page)

```

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    // This contract inherits the `onlyOwner` modifier from
    // `owned` and applies it to the `destroy` function, which
    // causes that calls to `destroy` only have an effect if
    // they are made by the stored owner.
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, destructible {
    mapping (address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }
}

```

(continues on next page)

(continued from previous page)

```

function changePrice(uint price_) public onlyOwner {
    price = price_;
}
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        ;
        locked = false;
    }
}

/// This function is protected by a mutex, which means that
/// reentrant calls from within `msg.sender.call` cannot call `f` again.
/// The `return 7` statement assigns 7 to the return value but still
/// executes the statement `locked = false` in the modifier.
function f() public noReentrancy returns (uint) {
    (bool success,) = msg.sender.call("");
    require(success);
    return 7;
}
}

```

If you want to access a modifier `m` defined in a contract `C`, you can use `C.m` to reference it without virtual lookup. It is only possible to use modifiers defined in the current contract or its base contracts. Modifiers can also be defined in libraries but their use is limited to functions of the same library.

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Modifiers cannot implicitly access or change the arguments and return values of functions they modify. Their values can only be passed to them explicitly at the point of invocation.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the `_` in the preceding modifier.

Warning: In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

An explicit return from a modifier with `return;` does not affect the values returned by the function. The modifier can, however, choose not to execute the function body at all and in that case the return variables are set to their *default values* just as if the function had an empty body.

The `_` symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

3.9.4 Constant and Immutable State Variables

State variables can be declared as `constant` or `immutable`. In both cases, the variables cannot be modified after the contract has been constructed. For `constant` variables, the value has to be fixed at compile-time, while for `immutable`, it can still be assigned at construction time.

It is also possible to define `constant` variables at the file level.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. For a `constant` variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations. `Immutable` variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, `constant` values can sometimes be cheaper than `immutable` values.

Not all types for constants and immutables are implemented at this time. The only supported types are `strings` (only for constants) and `value types`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint decimals_, address ref) {
        decimals = decimals_;
        // Assignments to immutables can even access the environment.
        maxBalance = ref.balance;
    }

    function isBalanceTooHigh(address other) public view returns (bool) {
        return other.balance > maxBalance;
    }
}
```

Constant

For `constant` variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. `block.timestamp`, `address(this).balance` or `block.number`) or execution data (`msg.value` or `gasleft()`) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` and `mulmod` are allowed (even though, with the exception of `keccak256`, they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

Immutable

Variables declared as `immutable` are a bit less restricted than those declared as `constant`: Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They can be assigned only once and can, from that point on, be read even during construction time.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables with the values assigned to them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

Note: Immutables that are assigned at their declaration are only considered initialized once the constructor of the contract is executing. This means you cannot initialize immutables inline with a value that depends on another immutable. You can do this, however, inside the constructor of the contract.

This is a safeguard against different interpretations about the order of state variable initialization and constructor execution, especially with regards to inheritance.

3.9.5 Functions

Functions can be defined inside and outside of contracts.

Functions outside of a contract, also called “free functions”, always have implicit `internal` *visibility*. Their code is included in all contracts that call them, similar to internal library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory arr) pure returns (uint s) {
    for (uint i = 0; i < arr.length; i++)
        s += arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory arr) public {
        // This calls the free function internally.
        // The compiler will add its code to the contract.
        uint s = sum(arr);
        require(s >= 10);
        found = true;
    }
}
```

Note: Functions defined outside a contract are still always executed in the context of a contract. They still have access to the variable `this`, can call other contracts, send them Ether and destroy the contract that called them, among other things. The main difference to functions defined inside a contract is that free functions do not have direct access to storage variables and functions not in their scope.

Function Parameters and Return Variables

Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

Function Parameters

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

For example, if you want your contract to accept one kind of external call with two integers, you would use something like the following:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    uint sum;
    function taker(uint a, uint b) public {
        sum = a + b;
    }
}
```

Function parameters can be used as any other local variable and they can also be assigned to.

Note: An *external function* cannot accept a multi-dimensional array as an input parameter. This functionality is possible if you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file.

An *internal function* can accept a multi-dimensional array without enabling the feature.

Return Variables

Function return variables are declared with the same syntax after the `returns` keyword.

For example, suppose you want to return two results: the sum and the product of two integers passed as function parameters, then you use something like:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        sum = a + b;
        product = a * b;
    }
}
```

The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their *default value* and have that value until they are (re-)assigned.

You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or *multiple ones*) directly with the `return` statement:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        return (a + b, a * b);
    }
}
```

If you use an early `return` to leave a function that has return variables, you must provide return values together with the `return` statement.

Note: You cannot return some types from non-internal functions, notably multi-dimensional dynamic arrays and structs. If you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file then more types are available, but mapping types are still limited to inside a single contract and you cannot transfer them.

Returning Multiple Values

When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an *implicit conversion*.

State Mutability

View Functions

Functions can be declared `view` in which case they promise not to modify the state.

Note: If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used when `view` functions are called, which enforces the state to stay unmodified as part of the EVM execution. For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`. This means library `view` functions do not have run-time checks that prevent state modifications. This should not impact security negatively because library code is usually known at compile-time and the static checker performs compile-time checks.

The following statements are considered modifying the state:

1. Writing to state variables.
2. *Emitting events.*
3. *Creating other contracts.*
4. Using `selfdestruct`.
5. Sending Ether via calls.

6. Calling any function not marked `view` or `pure`.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

Note: `constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

Note: Getter methods are automatically marked `view`.

Note: Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `view` functions. This enabled state modifications in `view` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `view` functions, modifications to the state are prevented on the level of the EVM.

Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state. In particular, it should be possible to evaluate a `pure` function at compile-time given only its inputs and `msg.data`, but without any knowledge of the current blockchain state. This means that reading from `immutable` variables can be a non-`pure` operation.

Note: If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used, which does not guarantee that the state is not read, but at least that it is not modified.

In addition to the list of state modifying statements explained above, the following are considered reading from the state:

1. Reading from state variables.
2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked `pure`.
5. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
```

(continues on next page)

(continued from previous page)

```

    return a * (b + 42);
}
}

```

Pure functions are able to use the `revert()` and `require()` functions to revert potential state changes when an [error occurs](#).

Reverting a state change is not considered a “state modification”, as only changes to the state made previously in code that did not have the `view` or `pure` restriction are reverted and that code has the option to catch the `revert` and not pass it on.

This behaviour is also in line with the `STATICCALL` opcode.

Warning: It is not possible to prevent functions from reading the state at the level of the EVM, it is only possible to prevent them from writing to the state (i.e. only `view` can be enforced at the EVM level, `pure` can not).

Note: Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for pure functions. This enabled state modifications in pure functions through the use of invalid explicit type conversions. By using `STATICCALL` for pure functions, modifications to the state are prevented on the level of the EVM.

Note: Prior to version 0.4.17 the compiler did not enforce that `pure` is not reading the state. It is a compile-time type check, which can be circumvented doing invalid explicit conversions between contract types, because the compiler can verify that the type of the contract does not do state-changing operations, but it cannot check that the contract that will be called at runtime is actually of that type.

Special Functions

Receive Ether Function

A contract can have at most one `receive` function, declared using `receive() external payable { ... }` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility and `payable` state mutability. It can be virtual, can override and can have modifiers.

The `receive` function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via `.send()` or `.transfer()`). If no such function exists, but a payable [fallback function](#) exists, the fallback function will be called on a plain Ether transfer. If neither a `receive` Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.

In the worst case, the `receive` function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging. The following operations will consume more gas than the 2300 gas stipend:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Warning: When Ether is sent directly to a contract (without a function call, i.e. sender uses `send` or `transfer`) but the receiving contract does not define a receive Ether function or a payable fallback function, an exception will be thrown, sending back the Ether (this was different before Solidity v0.4.0). If you want your contract to receive Ether, you have to implement a receive Ether function (using payable fallback functions for receiving Ether is not recommended, since the fallback is invoked and would not fail for interface confusions on the part of the sender).

Warning: A contract without a receive Ether function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

Below you can see an example of a Sink contract that uses function `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fallback Function

A contract can have at most one fallback function, declared using either `fallback () external [payable]` or `fallback (bytes calldata input) external [payable] returns (bytes memory output)` (both without the `function` keyword). This function must have `external` visibility. A fallback function can be virtual, can override and can have modifiers.

The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no *receive Ether function*. The fallback function always receives data, but in order to also receive Ether it must be marked `payable`.

If the version with parameters is used, `input` will contain the full data sent to the contract (equal to `msg.data`) and can return data in `output`. The returned data will not be ABI-encoded. Instead it will be returned without modifications (not even padding).

In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available (see *receive Ether function* for a brief description of the implications of this).

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

Warning: A payable fallback function is also executed for plain Ether transfers, if no *receive Ether function* is present. It is recommended to always define a receive Ether function as well, if you define a payable fallback function to distinguish Ether transfers from interface confusions.

Note: If you want to decode the input data, you can check the first four bytes for the function selector and then you can use `abi.decode` together with the array slice syntax to decode ABI-encoded data: `(c, d) = abi.decode(input[4:], [uint256, uint256]);` Note that this should only be used as a last resort and proper functions should be used instead.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the `payable`
    // modifier.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable { x = 1; y = msg.value; }

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
            "nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has no
        // payable
        // fallback function.
        // It has to be converted to the ``address payable`` type to even allow calling
        // ``send`` on it.
        address payable testPayable = payable(address(test));
    }
}
```

(continues on next page)

(continued from previous page)

```

    // If someone sends Ether to that contract,
    // the transfer will fail, i.e. this returns false here.
    return testPayable.send(2 ether);
}

function callTestPayable(TestPayable test) public returns (bool) {
    (bool success,) = address(test).call(abi.encodeWithSignature(
    →"nonExistingFunction()"));
    require(success);
    // results in test.x becoming == 1 and test.y becoming 0.
    (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
    →"nonExistingFunction()"));
    require(success);
    // results in test.x becoming == 1 and test.y becoming 1.

    // If someone sends Ether to that contract, the receive function in TestPayable
    →will be called.
    // Since that function writes to storage, it takes more gas than is available
    →with a
    // simple ``send`` or ``transfer``. Because of that, we have to use a low-level
    →call.
    (success,) = address(test).call{value: 2 ether}("");
    require(success);
    // results in test.x becoming == 2 and test.y becoming 2 ether.

    return true;
}
}

```

Function Overloading

A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading” and also applies to inherited functions. The following example shows overloading of the function `f` in the scope of contract A.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint value) public pure returns (uint out) {
        out = value;
    }

    function f(uint value, bool really) public pure returns (uint out) {
        if (really)
            out = value;
    }
}

```

Overloaded functions are also present in the external interface. It is an error if two externally visible functions differ by their Solidity types but not by their external types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// This will not compile
contract A {
    function f(B value) public pure returns (B out) {
        out = value;
    }

    function f(address value) public pure returns (address out) {
        out = value;
    }
}

contract B {
}
```

Both `f` function overloads above end up accepting the address type for the ABI although they are considered different inside Solidity.

Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

Note: Return parameters are not taken into account for overload resolution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint8 val) public pure returns (uint8 out) {
        out = val;
    }

    function f(uint256 val) public pure returns (uint256 out) {
        out = val;
    }
}
```

Calling `f(50)` would create a type error since `50` can be implicitly converted both to `uint8` and `uint256` types. On another hand `f(256)` would resolve to `f(uint256)` overload as `256` cannot be implicitly converted to `uint8`.

3.9.6 Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change with Serenity). The Log and its event data is not accessible from within contracts (not even from the contract that created them).

It is possible to request a Merkle proof for logs, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain. You have to supply block headers because the contract can only see the last 256 block hashes.

You can add the attribute `indexed` to up to three parameters which adds them to a special data structure known as “*topics*” instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a [reference type](#) for an indexed argument, the Keccak-256 hash of the value is stored as a topic instead.

All parameters without the `indexed` attribute are *ABI-encoded* into the data part of the log.

Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

For example, the code below uses the web3.js `subscribe("logs")` method to filter logs that match a topic with a certain address value:

The hash of the signature of the event is one of the topics, except if you declared the event with the `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name, you can only filter by the contract address. The advantage of anonymous events is that they are cheaper to deploy and call. It also allows you to declare four indexed arguments rather than three.

Note: Since the transaction log only stores the event data and not the type, you have to know the type of the event, including which parameter is indexed and if the event is anonymous in order to correctly interpret the data. In particular, it is possible to “fake” the signature of another event using an anonymous event.

Members of Events

- `event.selector`: For non-anonymous events, this is a `bytes32` value containing the keccak256 hash of the event signature, as used in the default topic.

Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed from,
        bytes32 indexed id,
        uint value
    );

    function deposit(bytes32 id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, id, msg.value);
    }
}
```

The use in the JavaScript API is as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var depositEvent = clientReceipt.Deposit();

// watch for changes
depositEvent.watch(function(error, result){
    // result contains non-indexed arguments and topics
    // given to the `Deposit` call.
    if (!error)
        console.log(result);
});

// Or pass a callback to start watching immediately
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

The output of the above looks like the following (trimmed):

```
{  
    "returnValues": {  
        "from": "0x1111...FFFFCCCC",  
        "id": "0x50...sd5adb20",  
        "value": "0x420042"  
    },  
    "raw": {  
        "data": "0x7f...91385",  
        "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]  
    }  
}
```

Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

3.9.7 Errors and the Revert Statement

Errors in Solidity provide a convenient and gas-efficient way to explain to the user why an operation failed. They can be defined inside and outside of contracts (including interfaces and libraries).

They have to be used together with the *revert statement* which causes all changes in the current call to be reverted and passes the error data back to the caller.

```
// SPDX-License-Identifier: GPL-3.0  
pragma solidity ^0.8.4;  
  
/// Insufficient balance for transfer. Needed `required` but only  
/// `available` available.  
/// @param available balance available.  
/// @param required requested amount to transfer.  
error InsufficientBalance(uint256 available, uint256 required);  
  
contract TestToken {  
    mapping(address => uint) balance;  
    function transfer(address to, uint256 amount) public {  
        if (amount > balance[msg.sender])  
            revert InsufficientBalance({  
                available: balance[msg.sender],  
                required: amount  
            });  
        balance[msg.sender] -= amount;  
        balance[to] += amount;  
    }  
    // ...  
}
```

Errors cannot be overloaded or overridden but are inherited. The same error can be defined in multiple places as long as the scopes are distinct. Instances of errors can only be created using `revert` statements.

The error creates data that is then passed to the caller with the revert operation to either return to the off-chain component or catch it in a *try/catch statement*. Note that an error can only be caught when coming from an external call, reverts happening in internal calls or inside the same function cannot be caught.

If you do not provide any parameters, the error only needs four bytes of data and you can use *NatSpec* as above to further explain the reasons behind the error, which is not stored on chain. This makes this a very cheap and convenient error-reporting feature at the same time.

More specifically, an error instance is ABI-encoded in the same way as a function call to a function of the same name and types would be and then used as the return data in the `revert` opcode. This means that the data consists of a 4-byte selector followed by *ABI-encoded* data. The selector consists of the first four bytes of the keccak256-hash of the signature of the error type.

Note: It is possible for a contract to revert with different errors of the same name or even with errors defined in different places that are indistinguishable by the caller. For the outside, i.e. the ABI, only the name of the error is relevant, not the contract or file where it is defined.

The statement `require(condition, "description");` would be equivalent to `if (!condition) revert Error("description")` if you could define error `Error(string)`. Note, however, that `Error` is a built-in type and cannot be defined in user-supplied code.

Similarly, a failing `assert` or similar conditions will revert with an error of the built-in type `Panic(uint256)`.

Note: Error data should only be used to give an indication of failure, but not as a means for control-flow. The reason is that the revert data of inner calls is propagated back through the chain of external calls by default. This means that an inner call can “forge” revert data that looks like it could have come from the contract that called it.

Members of Errors

- `error.selector`: A `bytes4` value containing the error selector.

3.9.8 Inheritance

Solidity supports multiple inheritance including polymorphism.

Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy. This has to be explicitly enabled on each function in the hierarchy using the `virtual` and `override` keywords. See [Function Overriding](#) for more details.

It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy (see below).

When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is compiled into the created contract. This means that all internal calls to functions of base contracts also just use internal function calls (`super.f(..)` will use JUMP and not a message call).

State variable shadowing is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

The general inheritance system is very similar to Python’s, especially concerning multiple inheritance, but there are also some *differences*.

Details are given in the following example.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Destructible is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}

abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}

// Multiple inheritance is possible. Note that `Owned` is
// also a base class of `Destructible`, yet there is only a single
// instance of `Owned` (as for virtual inheritance in C++).
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs. If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    // If you want the function to override, you need to use the
```

(continues on next page)

(continued from previous page)

```
// `override` keyword. You need to specify the `virtual` keyword again
// if you want this function to be overridden again.
function destroy() public virtual override {
    if (msg.sender == owner) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).unregister();
        // It is still possible to call a specific
        // overridden function.
        Destructible.destroy();
    }
}
}

// If a constructor takes an argument, it needs to be
// provided in the header or modifier-invocation-style at
// the constructor of the derived contract (see below).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Here, we only specify `override` and not `virtual`.
    // This means that contracts deriving from `PriceFeed`
    // cannot change the behaviour of `destroy` anymore.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Note that above, we call `Destructible.destroy()` to “forward” the destruction request. The way this is done is problematic, as seen in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.
    ~destroy(); }
}
```

(continues on next page)

(continued from previous page)

```
contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ Destructible.
    ↵destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

A call to `Final.destroy()` will call `Base2.destroy` because we specify it explicitly in the final override, but this function will bypass `Base1.destroy`. The way around this is to use `super`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

If `Base2` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base1.destroy()` (note that the final inheritance sequence is – starting with the most derived contract: `Final, Base2, Base1, Destructible, owned`). The actual function that is called when using `super` is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Function Overriding

Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header. The overriding function may only change the visibility of the overridden function from `external` to `public`. The mutability may be changed to a more strict one following the order: `nonpayable` can be overridden by `view` and `pure`. `view` can be overridden by `pure`. `payable` is an exception and cannot be changed to any other mutability.

The following example demonstrates changing mutability and visibility:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

For multiple inheritance, the most derived base contracts that define the same function must be specified explicitly after the `override` keyword. In other words, you have to specify all base contracts that define the same function and have not yet been overridden by another base contract (on some path through the inheritance graph). Additionally, if a contract inherits the same function from multiple (unrelated) bases, it has to explicitly override it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    function foo() public override(Base1, Base2) {}
}
```

An explicit override specifier is not required if the function is defined in a common base contract or if there is a unique function in a common base contract that already overrides all other functions.

```
// SPDX-License-Identifier: GPL-3.0
```

(continues on next page)

(continued from previous page)

```
pragma solidity >=0.6.0 <0.9.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// No explicit override required
contract D is B, C {}
```

More formally, it is not required to override a function (directly or indirectly) inherited from multiple bases if there is a base contract that is part of all override paths for the signature, and (1) that base implements the function and no paths from the current contract to the base mentions a function with that signature or (2) that base does not implement the function and there is at most one mention of the function in all paths from the current contract to that base.

In this sense, an override path for a signature is a path through the inheritance graph that starts at the contract under consideration and ends at a contract mentioning a function with that signature that does not override.

If you do not mark a function that overrides as `virtual`, derived contracts can no longer change the behaviour of that function.

Note: Functions with the `private` visibility cannot be `virtual`.

Note: Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

Note: Starting from Solidity 0.8.8, the `override` keyword is not required when overriding an interface function, except for the case where the function is defined in multiple bases.

Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A
{
    function f() external view virtual returns(uint) { return 5; }

contract B is A
{
    uint public override f;
}
```

Note: While public state variables can override external functions, they themselves cannot be overridden.

Modifier Overriding

Function modifiers can override each other. This works in the same way as [function overriding](#) (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

In case of multiple inheritance, all direct base contracts must be specified explicitly:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation, and where you can run contract initialisation code.

Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or their [default value](#) if you do not.

After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs additional gas linear to the length of the code. This code includes all functions that are part of the public interface and all functions that are reachable from there through function calls. It does not include the constructor code or internal functions that are only called from the constructor.

If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() {}`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

contract B is A(1) {
    constructor() {}
}
```

You can use internal parameters in a constructor (for example storage pointers). In this case, the contract has to be marked `abstract`, because these parameters cannot be assigned valid values from outside but only through the constructors of derived contracts.

Warning: Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

Warning: Prior to version 0.7.0, you had to specify the visibility of constructors as either `internal` or `public`.

Arguments for Base Constructors

The constructors of all the base contracts will be called following the linearization rules explained below. If the base constructors have arguments, derived contracts need to specify all of them. This can be done in two ways:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint x_) { x = x_; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() {}
}

// or through a "modifier" of the derived constructor...
contract Derived2 is Base {
    constructor(uint y) Base(y * y) {}
}

// or declare abstract...
```

(continues on next page)

(continued from previous page)

```
abstract contract Derived3 is Base {
}

// and have the next concrete derived contract initialize it.
contract DerivedFromDerived is Derived3 {
    constructor() Base(10 + 10) {}
}
```

One way is directly in the inheritance list (**is** Base(7)). The other is in the way a modifier is invoked as part of the derived constructor (Base(y * y)). The first way to do it is more convenient if the constructor argument is a constant and defines the behaviour of the contract or describes it. The second way has to be used if the constructor arguments of the base depend on those of the derived contract. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts' constructors, it must be declared abstract. In that case, when another contract derives from it, that other contract's inheritance list or constructor must provide the necessary parameters for all base classes that haven't had their parameters specified (otherwise, that other contract must be declared abstract as well). For example, in the above code snippet, see Derived3 and DerivedFromDerived.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the [Diamond Problem](#). Solidity is similar to Python in that it uses “C3 Linearization” to force a specific order in the directed acyclic graph (DAG) of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the **is** directive is important: You have to list the direct base contracts in the order from “most base-like” to “most derived”. Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error “Linearization of inheritance graph impossible”.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// This will not compile
contract C is A, X {}
```

The reason for this is that C requests X to override A (by specifying A, X in this order), but A itself requests to override X, which is a contradiction that cannot be resolved.

Due to the fact that you have to explicitly override a function that is inherited from multiple bases without a unique override, C3 linearization is not too important in practice.

One area where inheritance linearization is especially important and perhaps not as clear is when there are multiple constructors in the inheritance hierarchy. The constructors will always be executed in the linearized order, regardless of the order in which their arguments are provided in the inheriting contract's constructor. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

(continues on next page)

(continued from previous page)

```

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// Constructors are executed in the following order:
// 1 - Base1
// 2 - Base2
// 3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// Constructors are executed in the following order:
// 1 - Base2
// 2 - Base1
// 3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Constructors are still executed in the following order:
// 1 - Base2
// 2 - Base1
// 3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}

```

Inheriting Different Kinds of Members of the Same Name

It is an error when any of the following pairs in a contract have the same name due to inheritance:

- a function and a modifier
- a function and an event
- an event and a modifier

As an exception, a state variable getter can override an external function.

3.9.9 Abstract Contracts

Contracts must be marked as abstract when at least one of their functions is not implemented or when they do not provide arguments for all of their base contract constructors. Even if this is not the case, a contract may still be marked abstract, such as when you do not intend for the contract to be created directly. Abstract contracts are similar to [Interfaces](#) but an interface is more limited in what it can declare.

An abstract contract is declared using the `abstract` keyword as shown in the following example. Note that this contract needs to be defined as abstract, because the function `utterance()` is declared, but no implementation was provided (no implementation body `{ }` was given).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}
```

Such abstract contracts can not be instantiated directly. This is also true, if an abstract contract itself does implement all defined functions. The usage of an abstract contract as a base class is shown in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public pure virtual returns (bytes32);
}

contract Cat is Feline {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it needs to be marked as abstract as well.

Note that a function without implementation is different from a [Function Type](#) even though their syntax looks very similar.

Example of function without implementation (a function declaration):

```
function foo(address) external returns (address);
```

Example of a declaration of a variable whose type is a function type:

```
function(address) external returns (address) foo;
```

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the [Template method](#) and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say “any child of mine must implement this method”.

Note: Abstract contracts cannot override an implemented virtual function with an unimplemented one.

3.9.10 Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions:

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external in the interface, even if they are public in the contract.
- They cannot declare a constructor.
- They cannot declare state variables.
- They cannot declare modifiers.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

All functions declared in interfaces are implicitly `virtual` and any functions that override them do not need the `override` keyword. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

Interfaces can inherit from other interfaces. This has the same rules as normal inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Types defined inside interfaces and other contract-like structures can be accessed from other contracts: `Token`, `TokenType` or `Token.Coin`.

3.9.11 Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the DELEGATECALL (CALLCODE until Homestead) feature of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract, i.e. this points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise). Library functions can only be called directly (i.e. without the use of DELEGATECALL) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

Note: Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a *mechanism* that disallows state-modifying functions to be called directly (i.e. without DELEGATECALL).

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (using qualified access like `L.f()`). Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and types *stored in memory* will be passed by reference and not copied. To realize this in the EVM, the code of internal library functions that are called from a contract and all functions called from therein will at compile time be included in the calling contract, and a regular JUMP call will be used instead of a DELEGATECALL.

Note: The inheritance analogy breaks down when it comes to public functions. Calling a public library function with `L.f()` results in an external call (DELEGATECALL to be precise). In contrast, `A.f()` is an internal call when A is a base contract of the current contract.

The following example illustrates how to use libraries (but using a manual method, be sure to check out [using for](#) for a more advanced example to implement a set).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// We define a new struct datatype that will be used to
// hold its data in the calling contract.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // Note that the first parameter is of type "storage
    // reference" and thus only its storage address and not
    // its contents is passed as part of the call. This is a
    // special feature of library functions. It is idiomatic
    // to call the first parameter `self`, if the function can
    // be seen as a method of that object.
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])

```

(continues on next page)

(continued from previous page)

```

        return false; // already there
    self.flags[value] = true;
    return true;
}

function remove(Data storage self, uint value)
public
returns (bool)
{
    if (!self.flags[value])
        return false; // not there
    self.flags[value] = false;
    return true;
}

function contains(Data storage self, uint value)
public
view
returns (bool)
{
    return self.flags[value];
}
}

contract C {
    Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}

```

Of course, you do not have to follow this way to use libraries: they can also be used without defining struct data types. Functions also work without any storage reference parameters, and they can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (DELEGATECALL) to an external contract/library. If you use libraries, be aware that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of CALLCODE, `msg.sender` and `msg.value` changed, though).

The following example shows how to use *types stored in memory* and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {

```

(continues on next page)

(continued from previous page)

```

    uint[] limbs;
}

library BigInt {
    function fromUint(uint x) internal pure returns (bigint memory r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint memory a, bigint memory b) internal pure returns (bigint memory r) {
        r.limbs = new uint[](max(a.limbs.length, b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint limbA = limb(a, i);
            uint limbB = limb(b, i);
            unchecked {
                r.limbs[i] = limbA + limbB + carry;

                if (limbA + limbB < limbA || (limbA + limbB == type(uint).max && carry > 0))
                    carry = 1;
                else
                    carry = 0;
            }
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory a, uint index) internal pure returns (uint) {
        return index < a.limbs.length ? a.limbs[index] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
    }
}

```

(continues on next page)

(continued from previous page)

```

        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}

```

It is possible to obtain the address of a library by converting the library type to the `address` type, i.e. using `address(LibraryName)`.

As the compiler does not know the address where the library will be deployed, the compiled hex code will contain placeholders of the form `__$30bbc0abd4d6364515865950d3e0d10953$__`. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name, which would be for example `libraries/bigint.sol:BigInt` if the library was stored in a file called `bigint.sol` in a `libraries/` directory. Such bytecode is incomplete and should not be deployed. Placeholders need to be replaced with actual addresses. You can do that by either passing them to the compiler when the library is being compiled or by using the linker to update an already compiled binary. See [Library Linking](#) for information on how to use the commandline compiler for linking.

In comparison to contracts, libraries are restricted in the following ways:

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

(These might be lifted at a later point.)

Function Signatures and Selectors in Libraries

While external calls to public or external library functions are possible, the calling convention for such calls is considered to be internal to Solidity and not the same as specified for the regular [contract ABI](#). External library functions support more argument types than external contract functions, for example recursive structs and storage pointers. For that reason, the function signatures used to compute the 4-byte selector are computed following an internal naming schema and arguments of types not supported in the contract ABI use an internal encoding.

The following identifiers are used for the types in the signatures:

- Value types, non-storage `string` and non-storage `bytes` use the same identifiers as in the contract ABI.
- Non-storage array types follow the same convention as in the contract ABI, i.e. `<type>[]` for dynamic arrays and `<type>[M]` for fixed-size arrays of M elements.
- Non-storage structs are referred to by their fully qualified name, i.e. `C.S` for contract `C { struct S { ... } }`.
- Storage pointer mappings use `mapping(<keyType> => <valueType>) storage` where `<keyType>` and `<valueType>` are the identifiers for the key and value types of the mapping, respectively.
- Other storage pointer types use the type identifier of their corresponding non-storage type, but append a single space followed by `storage` to it.

The argument encoding is the same as for the regular contract ABI, except for storage pointers, which are encoded as a `uint256` value referring to the storage slot to which they point.

Similarly to the contract ABI, the selector consists of the first four bytes of the Keccak256-hash of the signature. Its value can be obtained from Solidity using the `.selector` member as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a CALL instead of a DELEGATECALL or CALLCODE, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using CALL or not, but a contract can use the ADDRESS opcode to find out “where” it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

This means that the actual code stored on chain for a library is different from the code reported by the compiler as `deployedBytecode`.

3.9.12 Using For

The directive `using A for B;` can be used to attach functions (A) as member functions to any type (B). These functions will receive the object they are called on as their first parameter (like the `self` variable in Python).

It is valid either at file level or inside a contract, at contract level.

The first part, A, can be one of:

- a list of file-level or library functions (`using {f, g, h, L.t} for uint;`) - only those functions will be attached to the type.
- the name of a library (`using L for uint;`) - all functions (both public and internal ones) of the library are attached to the type

At file level, the second part, B, has to be an explicit type (without data location specifier). Inside contracts, you can also use `using L for *;`, which has the effect that all functions of the library L are attached to *all* types.

If you specify a library, *all* functions in the library are attached, even those where the type of the first parameter does not match the type of the object. The type is checked at the point the function is called and function overload resolution is performed.

If you use a list of functions (`using {f, g, h, L.t} for uint;`), then the type (`uint`) has to be implicitly convertible to the first parameter of each of these functions. This check is performed even if none of these functions are called.

The `using A for B;` directive is active only within the current scope (either the contract or the current module/source unit), including within all of its functions, and has no effect outside of the contract or module in which it is used.

When the directive is used at file level and applied to a user-defined type which was defined at file level in the same file, the word `global` can be added at the end. This will have the effect that the functions are attached to the type everywhere the type is available (including other files), not only in the scope of the using statement.

Let us rewrite the set example from the [Libraries](#) section in this way, using file-level functions instead of library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

struct Data { mapping(uint => bool) flags; }
// Now we attach functions to the type.
// The attached functions can be used throughout the rest of the module.
// If you import the module, you have to
// repeat the using directive there, for example as
// import "flags.sol" as Flags;
// using {Flags.insert, Flags.remove, Flags.contains}
//      for Flags.Data;
using {insert, remove, contains} for Data;

function insert(Data storage self, uint value)
    returns (bool)
{
    if (self.flags[value])
        return false; // already there
    self.flags[value] = true;
    return true;
}

function remove(Data storage self, uint value)
    returns (bool)
{
    if (!self.flags[value])
        return false; // not there
    self.flags[value] = false;
    return true;
}

function contains(Data storage self, uint value)
    view
    returns (bool)
{
    return self.flags[value];
}

contract C {
    Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Data have
    }
}
```

(continues on next page)

(continued from previous page)

```
// corresponding member functions.
// The following function call is identical to
// `Set.insert(knownValues, value)`
require(knownValues.insert(value));
}
}
```

It is also possible to extend built-in types in that way. In this example, we will use a library.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}
using Search for uint[];

contract C {
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint from, uint to) public {
        // This performs the library function call
        uint index = data.indexOf(from);
        if (index == type(uint).max)
            data.push(to);
        else
            data[index] = to;
    }
}
```

Note that all external library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even in case of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used or when internal library functions are called.

3.10 Inline Assembly

You can interleave Solidity statements with inline assembly in a language close to the one of the Ethereum virtual machine. This gives you more fine-grained control, which is especially useful when you are enhancing the language by writing libraries.

The language used for inline assembly in Solidity is called *Yul* and it is documented in its own section. This section will only cover how the inline assembly code can interface with the surrounding Solidity code.

Warning: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

An inline assembly block is marked by `assembly { ... }`, where the code inside the curly braces is code in the *Yul* language.

The inline assembly code can access local Solidity variables as explained below.

Different inline assembly blocks share no namespace, i.e. it is not possible to call a Yul function or access a Yul variable defined in a different inline assembly block.

3.10.1 Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is possible with “plain Solidity” too, by using `<address>.code`. But the point here is that reusable assembly libraries can enhance the Solidity language without a compiler change.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library GetCode {
    function at(address addr) public view returns (bytes memory code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(addr)
            // allocate output byte array - this could also be done without assembly
            // by using code = new bytes(size)
            code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(addr, add(code, 0x20), 0, size)
        }
    }
}
```

Inline assembly is also beneficial in cases where the optimizer fails to produce efficient code, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;
```

(continues on next page)

(continued from previous page)

```

library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i)
            sum += data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory data) public pure returns (uint sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(data)

            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing data would result in an unusable
            //       data variable after this assembly block
            let dataElementLocation := add(data, 0x20)

            // Iterate until the bound is not met.
            for
                { let end := add(dataElementLocation, mul(len, 0x20)) }
                lt(dataElementLocation, end)
                { dataElementLocation := add(dataElementLocation, 0x20) }
            {
                sum := add(sum, mload(dataElementLocation))
            }
        }
    }
}

```

3.10.2 Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name.

Local variables of value type are directly usable in inline assembly. They can both be read and assigned to.

Local variables that refer to memory evaluate to the address of the variable in memory not the value itself. Such variables can also be assigned to, but note that an assignment will only change the pointer and not the data and that it is your responsibility to respect Solidity's memory management. See [Conventions in Solidity](#).

Similarly, local variables that refer to statically-sized calldata arrays or calldata structs evaluate to the address of the variable in calldata, not the value itself. The variable can also be assigned a new offset, but note that no validation to ensure that the variable will not point beyond `calldatasize()` is performed.

For external function pointers the address and the function selector can be accessed using `x.address` and `x.selector`. The selector consists of four right-aligned bytes. Both values can be assigned to. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Assigns a new selector and address to the return variable @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure
    -> returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address := newAddress
        }
    }
}
```

For dynamic calldata arrays, you can access their calldata offset (in bytes) and length (number of elements) using `x.offset` and `x.length`. Both expressions can also be assigned to, but as for the static case, no validation will be performed to ensure that the resulting data area is within the bounds of `calldatasize()`.

For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their “address” is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x.slot`, and to retrieve the byte-offset you use `x.offset`. Using `x` itself will result in an error.

You can also assign to the `.slot` part of a local storage variable pointer. For these (structs, arrays or mappings), the `.offset` part is always zero. It is not possible to assign to the `.slot` or `.offset` part of a state variable, though.

Local Solidity variables are available for assignments, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b.slot))
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}
```

Warning: If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, or `bytes16`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important: `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode: `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Since Solidity 0.6.0 the name of a inline assembly variable may not shadow any declaration visible in the scope of the inline assembly block (including variable, contract and function declarations).

Since Solidity 0.7.0, variables and functions declared inside the inline assembly block may not contain `.`, but using `.` is valid to access Solidity variables from outside the inline assembly block.

3.10.3 Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

3.10.4 Conventions in Solidity

Values of Typed Variables

In contrast to EVM assembly, Solidity has types which are narrower than 256 bits, e.g. `uint24`. For efficiency, most arithmetic operations ignore the fact that types can be shorter than 256 bits, and the higher-order bits are cleaned when necessary, i.e., shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher-order bits first.

Memory Management

Solidity manages memory in the following way. There is a “free memory pointer” at position `0x40` in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet you can use for allocating memory that follows the process outlined above

```

function allocate(length) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, length))
}
```

The first 64 bytes of memory can be used as “scratch space” for short-term allocation. The 32 bytes after the free memory pointer (i.e., starting at `0x60`) are meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Warning: Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically- and dynamically-sized arrays, so do not rely on this.

Memory Safety

Without the use of inline assembly, the compiler can rely on memory to remain in a well-defined state at all times. This is especially relevant for [the new code generation pipeline via Yul IR](#): this code generation path can move local variables from stack to memory to avoid stack-too-deep errors and perform additional memory optimizations, if it can rely on certain assumptions about memory use.

While we recommend to always respect Solidity's memory model, inline assembly allows you to use memory in an incompatible way. Therefore, moving stack variables to memory and additional memory optimizations are, by default, disabled in the presence of any inline assembly block that contains a memory operation or assigns to solidity variables in memory.

However, you can specifically annotate an assembly block to indicate that it in fact respects Solidity's memory model as follows:

```
assembly ("memory-safe") {  
    ...  
}
```

In particular, a memory-safe assembly block may only access the following memory ranges:

- Memory allocated by yourself using a mechanism like the `allocate` function described above.
- Memory allocated by Solidity, e.g. memory within the bounds of a memory array you reference.
- The scratch space between memory offset 0 and 64 mentioned above.
- Temporary memory that is located *after* the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is “allocated” at the free memory pointer without updating the free memory pointer.

Furthermore, if the assembly block assigns to Solidity variables in memory, you need to assure that accesses to the Solidity variables only access these memory ranges.

Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates. As an example, the following assembly snippet is not memory safe, because the value of `returnndatasize()` may exceed the 64 byte scratch space:

```
assembly {  
    returnndatacopy(0, 0, returnndatasize())  
    revert(0, returnndatasize())  
}
```

On the other hand, the following code *is* memory safe, because memory beyond the location pointed to by the free memory pointer can safely be used as temporary scratch space:

```
assembly ("memory-safe") {  
    let p := mload(0x40)  
    returnndatacopy(p, 0, returnndatasize())
```

(continues on next page)

(continued from previous page)

```
revert(p, returndatasize())
}
```

Note that you do not need to update the free memory pointer if there is no following allocation, but you can only use memory starting from the current offset given by the free memory pointer.

If the memory operations use a length of zero, it is also fine to just use any offset (not only if it falls into the scratch space):

```
assembly ("memory-safe") {
    revert(0, 0)
}
```

Note that not only memory operations in inline assembly itself can be memory-unsafe, but also assignments to solidity variables of reference type in memory. For example the following is not memory-safe:

```
bytes memory x;
assembly {
    x := 0x40
}
x[0x20] = 0x42;
```

Inline assembly that neither involves any operations that access memory nor assigns to any solidity variables in memory is automatically considered memory-safe and does not need to be annotated.

Warning: It is your responsibility to make sure that the assembly actually satisfies the memory model. If you annotate an assembly block as memory-safe, but violate one of the memory assumptions, this **will** lead to incorrect and undefined behaviour that cannot easily be discovered by testing.

In case you are developing a library that is meant to be compatible across multiple versions of solidity, you can use a special comment to annotate an assembly block as memory-safe:

```
/// @solidity memory-safe-assembly
assembly {
    ...
}
```

Note that we will disallow the annotation via comment in a future breaking release, so if you are not concerned with backwards-compatibility with older compiler versions, prefer using the dialect string.

3.11 Cheatsheet

3.11.1 Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	<code>++, --</code>
	New expression	<code>new <typename></code>
	Array subscripting	<code><array>[<index>]</code>
	Member access	<code><object>.⟨member⟩</code>
	Function-like call	<code><func>(<args...>)</code>
	Parentheses	<code>(⟨statement⟩)</code>
2	Prefix increment and decrement	<code>++, --</code>
	Unary minus	<code>-</code>
	Unary operations	<code>delete</code>
	Logical NOT	<code>!</code>
	Bitwise NOT	<code>~</code>
3	Exponentiation	<code>**</code>
4	Multiplication, division and modulo	<code>*, /, %</code>
5	Addition and subtraction	<code>+, -</code>
6	Bitwise shift operators	<code><<, >></code>
7	Bitwise AND	<code>&</code>
8	Bitwise XOR	<code>^</code>
9	Bitwise OR	<code> </code>
10	Inequality operators	<code><, >, <=, >=</code>
11	Equality operators	<code>==, !=</code>
12	Logical AND	<code>&&</code>
13	Logical OR	<code> </code>
14	Ternary operator	<code><conditional> ? <if-true> : <if-false></code>
	Assignment operators	<code>=, =, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=</code>
15	Comma operator	<code>,</code>

3.11.2 Global Variables

- `abi.decode(bytes memory encodedData, (...)) returns (...):` *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...)` returns `(bytes memory)`: *ABI*-encodes the given arguments
- `abi.encodePacked(...)` returns `(bytes memory)`: Performs *packed encoding* of the given arguments. Note that this encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...)` returns `(bytes memory)`: *ABI*-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeCall(function functionPointer, (...))` returns `(bytes memory)`: ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, (...))`
- `abi.encodeWithSignature(string memory signature, ...)` returns `(bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `bytes.concat(...)` returns `(bytes memory)`: *Concatenates variable number of arguments to one byte array*
- `string.concat(...)` returns `(string memory)`: *Concatenates variable number of arguments to one string array*

- `block.basefee(uint)`: current block's base fee ([EIP-3198](#) and [EIP-1559](#))
- `block.chainid(uint)`: current chain id
- `block.coinbase(address payable)`: current block miner's address
- `block.difficulty(uint)`: current block difficulty
- `block.gaslimit(uint)`: current block gaslimit
- `block.number(uint)`: current block number
- `block.timestamp(uint)`: current block timestamp in seconds since Unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data(bytes)`: complete calldata
- `msg.sender(address)`: sender of the message (current call)
- `msg.sig(bytes4)`: first four bytes of the calldata (i.e. function identifier)
- `msg.value(uint)`: number of wei sent with the message
- `tx.gasprice(uint)`: gas price of the transaction
- `tx.origin(address)`: sender of the transaction (full call chain)
- `assert(bool condition)`: abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory message)`: abort execution and revert state changes providing an explanatory string
- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input
- `sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input
- `ripemd160(bytes memory) returns (bytes20)`: compute the RIPEMD-160 hash of the input
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)`: compute $(x + y) \% k$ where the addition is performed with arbitrary precision and does not wrap around at 2^{**256} . Assert that $k \neq 0$ starting from version 0.5.0.
- `mulmod(uint x, uint y, uint k) returns (uint)`: compute $(x * y) \% k$ where the multiplication is performed with arbitrary precision and does not wrap around at 2^{**256} . Assert that $k \neq 0$ starting from version 0.5.0.
- `this` (current contract's type): the current contract, explicitly convertible to `address` or `address payable`
- `super`: the contract one level higher in the inheritance hierarchy
- `selfdestruct(address payable recipient)`: destroy the current contract, sending its funds to the given address

- <address>.balance (uint256): balance of the *Address* in Wei
- <address>.code (bytes memory): code at the *Address* (can be empty)
- <address>.codehash (bytes32): the codehash of the *Address*
- <address payable>.send(uint256 amount) returns (bool): send given amount of Wei to *Address*, returns false on failure
- <address payable>.transfer(uint256 amount): send given amount of Wei to *Address*, throws on failure
- type(C).name (string): the name of the contract
- type(C).creationCode (bytes memory): creation bytecode of the given contract, see *Type Information*.
- type(C).runtimeCode (bytes memory): runtime bytecode of the given contract, see *Type Information*.
- type(I).interfaceId (bytes4): value containing the EIP-165 interface identifier of the given interface, see *Type Information*.
- type(T).min (T): the minimum value representable by the integer type T, see *Type Information*.
- type(T).max (T): the maximum value representable by the integer type T, see *Type Information*.

3.11.3 Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- **public**: visible externally and internally (creates a *getter function* for storage/state variables)
- **private**: only visible in the current contract
- **external**: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- **internal**: only visible internally

3.11.4 Modifiers

- **pure** for functions: Disallows modification or access of state.
- **view** for functions: Disallows modification of state.
- **payable** for functions: Allows them to receive Ether together with a call.
- **constant** for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- **immutable** for state variables: Allows exactly one assignment at construction time and is constant afterwards. Is stored in code.
- **anonymous** for events: Does not store event signature as topic.
- **indexed** for event parameters: Stores the parameter as topic.
- **virtual** for functions and modifiers: Allows the function's or modifier's behaviour to be changed in derived contracts.
- **override**: States that this function, modifier or public state variable changes the behaviour of a function or modifier in a base contract.

3.11.5 Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

after, alias, apply, auto, byte, case, copyof, default, define, final, implements, in, inline, let, macro, match, mutable, null, of, partial, promise, reference, relocatable, sealed, sizeof, static, supports, switch, typedef, typeof, var.

3.12 Using the Compiler

3.12.1 Using the Commandline Compiler

Note: This section does not apply to `solcjs`, not even if it is used in commandline mode.

Basic Usage

One of the build targets of the Solidity repository is `solc`, the solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol`.

Optimizer Options

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime (more specifically, it assumes each opcode is executed around 200 times). If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number. This parameter has effects on the following (this might change in the future):

- the size of the binary search in the function dispatch routine
- the way constants like large numbers or strings are stored

Base Path and Import Remapping

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide *path redirects* using `prefix=path` in the following way:

```
solc github.com/ethereum/dapp-bin/=~/usr/local/lib/dapp-bin/ file.sol
```

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`.

When accessing the filesystem to search for imports, *paths that do not start with `./` or `../`* are treated as relative to the directories specified using `--base-path` and `--include-path` options (or the current working directory if base path is not specified). Furthermore, the part of the path added via these options will not appear in the contract metadata.

For security reasons the compiler has *restrictions on what directories it can access*. Directories of source files specified on the command line and target paths of remappings are automatically allowed to be accessed by the file reader, but everything else is rejected by default. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch. Everything inside the path specified via `--base-path` is always allowed.

The above is only a simplification of how the compiler handles import paths. For a detailed explanation with examples and discussion of corner cases please refer to the section on *path resolution*.

Library Linking

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__$53aea86b7d70b31448b230b20ae141a537$__`. These are placeholders for the actual library addresses. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name. The bytecode file will also contain lines of the form `// <placeholder> -> <fq library name>` at the end to help identify which libraries the placeholders represent. Note that the fully qualified library name is the path of its source file and the library name separated by `:`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points:

Either add `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD567890123456789012345678901234567890"` to your command to provide an address for each library (use commas or spaces as separators) or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

Note: Starting Solidity 0.8.1 accepts `=` as separator between library and address, and `:` as a separator is deprecated. It will be removed in the future. Currently `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD567890123456789012345678901234567890` will work too.

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses. The process will always terminate in a “success” state and report any errors via the JSON output. The option `--base-path` is also processed in standard-json mode.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__$53aea86b7d70b31448b230b20ae141a537$__`-format given above and are linked in-place (if the input is read from `stdin`, it is written to `stdout`). All options except `--libraries` are ignored (including `-o`) in this case.

Warning: Manually linking libraries on the generated bytecode is discouraged because it does not update contract metadata. Since metadata contains a list of libraries specified at the time of compilation and bytecode contains a metadata hash, you will get different binaries, depending on when linking is performed.

You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

Note: The library placeholder used to be the fully qualified name of the library itself instead of the hash of it. This format is still supported by `solc --link` but the compiler will no longer output it. This change was made to reduce the likelihood of a collision between libraries, since only the first 36 characters of the fully qualified library name could be used.

3.12.2 Setting the EVM Version to Target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

Warning: Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command line, you can select the EVM version as follows:

```
solc --evm-version <VERSION> contract.sol
```

In the *standard JSON interface*, use the "evmVersion" key in the "settings" field:

```
{
  "sources": {/* ... */},
  "settings": {
    "optimizer": {/* ... */},
    "evmVersion": "<VERSION>"
  }
}
```

Target Options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- **homestead**
 - (oldest version)
- **tangerineWhistle**
 - Gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
 - All gas sent by default for external calls, previously a certain amount had to be retained.
- **spuriousDragon**
 - Gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- **byzantium**
 - Opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
 - The `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
 - It is possible to access dynamic data returned from function calls.
 - `revert` opcode introduced, which means that `revert()` will not waste gas.
- **constantinople**
 - Opcodes `create2``, `extcodehash`, `shl`, `shr` and `sar` are available in assembly.
 - Shifting operators use shifting opcodes and thus need less gas.
- **petersburg**

- The compiler behaves the same way as with constantinople.
- **istanbul**
 - Opcodes `chainid` and `selfbalance` are available in assembly.
- **berlin**
 - Gas costs for `SLOAD`, `*CALL`, `BALANCE`, `EXT*` and `SELFDESTRUCT` increased. The compiler assumes cold gas costs for such operations. This is relevant for gas estimation and the optimizer.
- **london (default)**
 - The block's base fee ([EIP-3198](#) and [EIP-1559](#)) can be accessed via the global `block.basefee` or `basefee()` in inline assembly.

3.12.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The standard error output is not used and the process will always terminate in a “success” state, even if there were errors. Errors are always reported as part of the JSON output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

Input Description

```
{  
    // Required: Source code language. Currently supported are "Solidity" and "Yul".  
    "language": "Solidity",  
    // Required  
    "sources":  
    {  
        // The keys here are the "global" names of the source files,  
        // imports can use other files via remappings (see below).  
        "myFile.sol":  
        {  
            // Optional: keccak256 hash of the source file  
            // It is used to verify the retrieved content if imported via URLs.  
            "keccak256": "0x123...",  
            // Required (unless "content" is used, see below): URL(s) to the source file.  
            // URL(s) should be imported in this order and the result checked against the  
            // keccak256 hash (if available). If the hash doesn't match or none of the  
            // URL(s) result in success, an error should be raised.  
            // Using the commandline interface only filesystem paths are supported.  
            // With the JavaScript interface the URL will be passed to the user-supplied  
            // read callback, so any URL supported by the callback can be used.  
            "urls":  
            [  
                "bzzr://56ab...",
```

(continues on next page)

(continued from previous page)

```

"ipfs://Qma...",
"/tmp/path/to/file.sol"
// If files are used, their directories should be added to the command line via
// `--allow-paths <path>`.

],
{
  "destructible": {
    // Optional: keccak256 hash of the source file
    "keccak256": "0x234...",
    // Required (unless "urls" is used): literal contents of the source file
    "content": "contract destructible is owned { function shutdown() { if (msg.sender==owner) selfdestruct(owner); } }"
  }
},
// Optional
"settings": {
  // Optional: Stop compilation after the given stage. Currently only "parsing" is valid here
  "stopAfter": "parsing",
  // Optional: Sorted list of remappings
  "remappings": [ ":g=/dir" ],
  // Optional: Optimizer settings
  "optimizer": {
    // Disabled by default.
    // NOTE: enabled=false still leaves some optimizations on. See comments below.
    // WARNING: Before version 0.8.6 omitting the 'enabled' key was not equivalent to setting it to false and would actually disable all the optimizations.
    "enabled": true,
    // Optimize for how many times you intend to run the code.
    // Lower values will optimize more for initial deployment cost, higher values will optimize more for high-frequency usage.
    "runs": 200,
    // Switch optimizer components on or off in detail.
    // The "enabled" switch above provides two defaults which can be tweaked here. If "details" is given, "enabled" can be omitted.
    "details": {
      // The peephole optimizer is always on if no details are given, use details to switch it off.
      "peephole": true,
      // The inliner is always on if no details are given, use details to switch it off.
      "inliner": true,
      // The unused jumpdest remover is always on if no details are given, use details to switch it off.
      "jumpdestRemover": true,
      // Sometimes re-orders literals in commutative operations.
      "orderLiterals": false,
      // Removes duplicate code blocks
      "deduplicate": false,
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

// Common subexpression elimination, this is the most complicated step but
// can also provide the largest gain.
"cse": false,
// Optimize representation of literal numbers and strings in code.
"constantOptimizer": false,
// The new Yul optimizer. Mostly operates on the code of ABI coder v2
// and inline assembly.
// It is activated together with the global optimizer setting
// and can be deactivated here.
// Before Solidity 0.6.0 it had to be activated through this switch.
"yul": false,
// Tuning options for the Yul optimizer.
"yulDetails": {
    // Improve allocation of stack slots for variables, can free up stack slots.
    ↵early.
    // Activated by default if the Yul optimizer is activated.
    "stackAllocation": true,
    // Select optimization steps to be applied.
    // Optional, the optimizer will use the default sequence if omitted.
    "optimizerSteps": "dhfoDgvulfnTUtnIf..."
}
},
// Version of the EVM to compile for.
// Affects type checking and code generation. Can be homestead,
// tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul,
// or berlin
"evmVersion": "byzantium",
// Optional: Change compilation pipeline to go through the Yul intermediate.
// representation.
// This is false by default.
"viaIR": true,
// Optional: Debugging settings
"debug": {
    // How to treat revert (and require) reason strings. Settings are
    // "default", "strip", "debug" and "verboseDebug".
    // "default" does not inject compiler-generated revert strings and keeps user-
    // supplied ones.
    // "strip" removes all revert strings (if possible, i.e. if literals are used).
    // keeping side-effects
    // "debug" injects strings for compiler-generated internal reverts, implemented
    // for ABI encoders V1 and V2 for now.
    // "verboseDebug" even appends further information to user-supplied revert strings.
    // (not yet implemented)
    "revertStrings": "default",
    // Optional: How much extra debug information to include in comments in the
    // produced EVM
    // assembly and Yul code. Available components are:
    // - `location`: Annotations of the form `@src <index>:<start>:<end>` indicating the
    //   location of the corresponding element in the original Solidity file, where:
    //   - `<index>` is the file index matching the `@use-src` annotation,
    //   - `<start>` is the index of the first byte at that location,

```

(continues on next page)

(continued from previous page)

```

//      - `<end>` is the index of the first byte after that location.
// - `snippet`: A single-line code snippet from the location indicated by `@src`.
//      The snippet is quoted and follows the corresponding `@src` annotation.
// - `*`: Wildcard value that can be used to request everything.
"debugInfo": ["location", "snippet"]
},
// Metadata settings (optional)
"metadata": {
    // Use only literal content and not URLs (false by default)
    "useLiteralContent": true,
    // Use the given hash method for the metadata hash that is appended to the
    // bytecode.
    // The metadata hash can be removed from the bytecode via option "none".
    // The other options are "ipfs" and "bzzr1".
    // If the option is omitted, "ipfs" is used by default.
    "bytecodeHash": "ipfs"
},
// Addresses of the libraries. If not all libraries are given here,
// it can result in unlinked objects whose output data is different.
"libraries": {
    // The top level key is the name of the source file where the library is used.
    // If remappings are used, this source file should match the global path
    // after remappings were applied.
    // If this key is an empty string, that refers to a global level.
    "myFile.sol": {
        "MyLib": "0x123123..."
    }
},
// The following can be used to select desired outputs based
// on file and contract names.
// If this field is omitted, then the compiler loads and does type checking,
// but will not generate any outputs apart from errors.
// The first level key is the file name and the second level key is the contract
// name.
// An empty contract name is used for outputs that are not tied to a contract
// but to the whole source file like the AST.
// A star as contract name refers to all contracts in the file.
// Similarly, a star as a file name matches all files.
// To select all outputs the compiler can possibly generate, use
// "outputSelection: { "*": { "*": [ "*" ],"": [ "*" ] } }"
// but note that this might slow down the compilation process needlessly.
//
// The available output types are as follows:
//
// File level (needs empty string as contract name):
//   ast - AST of all source files
//
// Contract level (needs the contract name or "*"):
//   abi - ABI
//   devdoc - Developer documentation (natspec)
//   userdoc - User documentation (natspec)
//   metadata - Metadata

```

(continues on next page)

(continued from previous page)

```

// ir - Yul intermediate representation of the code before optimization
// irOptimized - Intermediate representation after optimization
// storageLayout - Slots, offsets and types of the contract's state variables.
// evm.assembly - New assembly format
// evm.legacyAssembly - Old-style assembly format in JSON
// evm.bytecode.functionDebugData - Debugging information at function level
// evm.bytecode.object - Bytecode object
// evm.bytecode.opcodes - Opcodes list
// evm.bytecode.sourceMap - Source mapping (useful for debugging)
// evm.bytecode.linkReferences - Link references (if unlinked object)
// evm.bytecode.generatedSources - Sources generated by the compiler
// evm.deployedBytecode* - Deployed bytecode (has all the options that evm.
↳bytecode has)
    // evm.deployedBytecode.immutableReferences - Map from AST ids to bytecode ranges
↳that reference immutables
    // evm.methodIdentifiers - The list of function hashes
    // evm.gasEstimates - Function gas estimates
    // ewasm.wast - Ewasm in WebAssembly S-expressions format
    // ewasm.wasm - Ewasm in WebAssembly binary format
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to request
↳everything.
    //
"outputSelection": {
    "*": {
        "*": [
            "metadata", "evm.bytecode" // Enable the metadata and bytecode outputs of
↳every single contract.
            , "evm.bytecode.sourceMap" // Enable the source map output of every single
↳contract.
        ],
        """: [
            "ast" // Enable the AST output of every single file.
        ]
    },
    // Enable the abi and opcodes output of MyContract defined in file def.
    "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
    }
},
// The modelChecker object is experimental and subject to changes.
"modelChecker":
{
    // Choose which contracts should be analyzed as the deployed one.
    "contracts":
    {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
    },
    // Choose how division and modulo operations should be encoded.
    // When using `false` they are replaced by multiplication with slack

```

(continues on next page)

(continued from previous page)

```

// variables. This is the default.
// Using `true` here is recommended if you are using the CHC engine
// and not using Spacer as the Horn solver (using Eldarica, for example).
// See the Formal Verification section for a more detailed explanation of this
// option.
    "divModNoSlacks": false,
    // Choose which model checker engine to use: all (default), bmc, chc, none.
    "engine": "chc",
    // Choose which types of invariants should be reported to the user: contract,
    // reentrancy.
    "invariants": ["contract", "reentrancy"],
    // Choose whether to output all unproved targets. The default is `false`.
    "showUnproved": true,
    // Choose which solvers should be used, if available.
    // See the Formal Verification section for the solvers description.
    "solvers": ["cvc4", "smtlib2", "z3"],
    // Choose which targets should be checked: constantCondition,
    // underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
    // If the option is not given all targets are checked by default,
    // except underflow/overflow for Solidity >=0.8.7.
    // See the Formal Verification section for the targets description.
    "targets": ["underflow", "overflow", "assert"],
    // Timeout for each SMT query in milliseconds.
    // If this option is not given, the SMTChecker will use a deterministic
    // resource limit by default.
    // A given timeout of 0 means no resource/time restrictions for any query.
    "timeout": 20000
}
}
}

```

Output Description

```

{
    // Optional: not present if no errors/warningsinfos were encountered
    "errors": [
        {
            // Optional: Location within the source file.
            "sourceLocation": {
                "file": "sourceFile.sol",
                "start": 0,
                "end": 100
            },
            // Optional: Further locations (e.g. places of conflicting declarations)
            "secondarySourceLocations": [
                {
                    "file": "sourceFile.sol",
                    "start": 64,
                    "end": 92,
                    "message": "Other declaration is here:"
                }
            ]
        }
    ]
}

```

(continues on next page)

(continued from previous page)

```

        }
    ],
    // Mandatory: Error type, such as "TypeError", "InternalCompilerError", "Exception",
    // etc.
    // See below for complete list of types.
    "type": "TypeError",
    // Mandatory: Component where the error originated, such as "general", "ewasm", etc.
    "component": "general",
    // Mandatory ("error", "warning" or "info", but please note that this may be
    // extended in the future)
    "severity": "error",
    // Optional: unique code for the cause of the error
    "errorCode": "3141",
    // Mandatory
    "message": "Invalid keyword",
    // Optional: the message formatted with source location
    "formattedMessage": "sourceFile.sol:100: Invalid keyword"
}
],
// This contains the file-level outputs.
// It can be limited/filtered by the outputSelection settings.
"sources": {
    "sourceFile.sol": {
        // Identifier of the source (used in source maps)
        "id": 1,
        // The AST object
        "ast": {}
    }
},
// This contains the contract-level outputs.
// It can be limited/filtered by the outputSelection settings.
"contracts": {
    "sourceFile.sol": {
        // If the language used has no contract names, this field should equal to an empty
        // string.
        "ContractName": {
            // The Ethereum Contract ABI. If empty, it is represented as an empty array.
            // See https://docs.soliditylang.org/en/develop/abi-spec.html
            "abi": [],
            // See the Metadata Output documentation (serialised JSON string)
            "metadata": "/* ... */",
            // User documentation (natspec)
            "userdoc": {},
            // Developer documentation (natspec)
            "devdoc": {},
            // Intermediate representation (string)
            "ir": "",
            // See the Storage Layout documentation.
            "storageLayout": {"storage": /* ... */[], "types": /* ... */{} },
            // EVM-related outputs
            "evm": {

```

(continues on next page)

(continued from previous page)

```

// Assembly (string)
"assembly": "",
// Old-style assembly (object)
"legacyAssembly": {},
// Bytecode and related details.
"bytecode": {
    // Debugging data at the level of functions.
    "functionDebugData": {
        // Now follows a set of functions including compiler-internal and
        // user-defined function. The set does not have to be complete.
        "@mint_13": { // Internal name of the function
            "entryPoint": 128, // Byte offset into the bytecode where the function starts (optional)
            "id": 13, // AST ID of the function definition or null for compiler-internal functions (optional)
            "parameterSlots": 2, // Number of EVM stack slots for the function parameters (optional)
            "returnSlots": 1 // Number of EVM stack slots for the return values (optional)
        }
    },
    // The bytecode as a hex string.
    "object": "00fe",
    // Opcodes list (string)
    "opcodes": "",
    // The source mapping as a string. See the source mapping definition.
    "sourceMap": "",
    // Array of sources generated by the compiler. Currently only
    // contains a single Yul file.
    "generatedSources": [
        // Yul AST
        "ast": {/* ... */},
        // Source file in its text form (may contain comments)
        "contents": "{ function abi_decode(start, end) -> data { data := calldataload(start) } }",
        // Source file ID, used for source references, same "namespace" as the Solidity source files
        "id": 2,
        "language": "Yul",
        "name": "#utility.yul"
    ],
    // If given, this is an unlinked object.
    "linkReferences": {
        "libraryFile.sol": {
            // Byte offsets into the bytecode.
            // Linking replaces the 20 bytes located there.
            "Library1": [
                { "start": 0, "length": 20 },
                { "start": 200, "length": 20 }
            ]
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
        },
        "deployedBytecode": {
            /* ..., */ // The same layout as above.
            "immutableReferences": {
                // There are two references to the immutable with AST ID 3, both 32 bytes
→long. One is
                // at bytecode offset 42, the other at bytecode offset 80.
                "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
            }
        },
        // The list of function hashes
        "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
        },
        // Function gas estimates
        "gasEstimates": {
            "creation": {
                "codeDepositCost": "420000",
                "executionCost": "infinite",
                "totalCost": "infinite"
            },
            "external": {
                "delegate(address)": "25000"
            },
            "internal": {
                "heavyLifting()": "infinite"
            }
        }
    },
    // Ewasm related outputs
    "ewasm": {
        // S-expressions format
        "wast": "",
        // Binary format (hex string)
        "wasm": ""
    }
}
}
```

Error Types

1. **JSONError**: JSON input doesn't conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
 2. **IOError**: IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
 3. **ParserError**: Source code doesn't conform to the language rules.
 4. **DocstringParsingError**: The NatSpec tags in the comment block cannot be parsed.
 5. **SyntaxError**: Syntactical error, such as `continue` is used outside of a `for` loop.

6. **DeclarationError**: Invalid, unresolvable or clashing identifier names. e.g. Identifier not found
7. **TypeError**: Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. **UnimplementedFeatureError**: Feature is not supported by the compiler, but is expected to be supported in future versions.
9. **InternalCompilerError**: Internal bug triggered in the compiler - this should be reported as an issue.
10. **Exception**: Unknown failure during compilation - this should be reported as an issue.
11. **CompilerError**: Invalid use of the compiler stack - this should be reported as an issue.
12. **FatalError**: Fatal error not processed correctly - this should be reported as an issue.
13. **Warning**: A warning, which didn't stop the compilation, but should be addressed if possible.
14. **Info**: Information that the compiler thinks the user might find useful, but is not dangerous and does not necessarily need to be addressed.

3.12.4 Compiler Tools

solidity-upgrade

`solidity-upgrade` can help you to semi-automatically upgrade your contracts to breaking language changes. While it does not and cannot implement all required changes for every breaking release, it still supports the ones, that would need plenty of repetitive manual adjustments otherwise.

Note: `solidity-upgrade` carries out a large part of the work, but your contracts will most likely need further manual adjustments. We recommend using a version control system for your files. This helps reviewing and eventually rolling back the changes made.

Warning: `solidity-upgrade` is not considered to be complete or free from bugs, so please use with care.

How it Works

You can pass (a) Solidity source file(s) to `solidity-upgrade [files]`. If these make use of `import` statement which refer to files outside the current source file's directory, you need to specify directories that are allowed to read and import files from, by passing `--allow-paths [directory]`. You can ignore missing files by passing `--ignore-missing`. `solidity-upgrade` is based on `libsolidity` and can parse, compile and analyse your source files, and might find applicable source upgrades in them.

Source upgrades are considered to be small textual changes to your source code. They are applied to an in-memory representation of the source files given. The corresponding source file is updated by default, but you can pass `--dry-run` to simulate the whole upgrade process without writing to any file.

The upgrade process itself has two phases. In the first phase source files are parsed, and since it is not possible to upgrade source code on that level, errors are collected and can be logged by passing `--verbose`. No source upgrades available at this point.

In the second phase, all sources are compiled and all activated upgrade analysis modules are run alongside compilation. By default, all available modules are activated. Please read the documentation on [available modules](#) for further details.

This can result in compilation errors that may be fixed by source upgrades. If no errors occur, no source upgrades are being reported and you're done. If errors occur and some upgrade module reported a source upgrade, the first reported one gets applied and compilation is triggered again for all given source files. The previous step is repeated as long as source upgrades are reported. If errors still occur, you can log them by passing `--verbose`. If no errors occur, your contracts are up to date and can be compiled with the latest version of the compiler.

Available Upgrade Modules

Module	Version	Description
<code>constructor</code>	0.5.0	Constructors must now be defined using the <code>constructor</code> keyword.
<code>visibility</code>	0.5.0	Explicit function visibility is now mandatory, defaults to <code>public</code> .
<code>abstract</code>	0.6.0	The keyword <code>abstract</code> has to be used if a contract does not implement all its functions.
<code>virtual</code>	0.6.0	Functions without implementation outside an interface have to be marked <code>virtual</code> .
<code>override</code>	0.6.0	When overriding a function or modifier, the new keyword <code>override</code> must be used.
<code>dotsyntax</code>	0.7.0	The following syntax is deprecated: <code>f.gas(...)(...)</code> , <code>f.value(...)(...)</code> and <code>(new C).value(...)(...)</code> . Replace these calls by <code>f{gas: ..., value: ...}()</code> and <code>(new C){value: ...}()</code> .
<code>now</code>	0.7.0	The <code>now</code> keyword is deprecated. Use <code>block.timestamp</code> instead.
<code>constructor</code>	visibility	Specifies visibility of constructors.

Please read [0.5.0 release notes](#), [0.6.0 release notes](#), [0.7.0 release notes](#) and [0.8.0 release notes](#) for further details.

Synopsis

```
Usage: solidity-upgrade [options] contract.sol
```

Allowed options:

- `--help` Show help message and exit.
- `--version` Show version and exit.
- `--allow-paths path(s)` Allow a given path for imports. A list of paths can be supplied by separating them with a comma.
- `--ignore-missing` Ignore missing files.
- `--modules module(s)` Only activate a specific upgrade module. A list of modules can be supplied by separating them with a comma.
- `--dry-run` Apply changes in-memory only and don't write to input file.
- `--verbose` Print logs, errors and changes. Shortens output of upgrade patches.
- `--unsafe` Accept *unsafe* changes.

Bug Reports / Feature Requests

If you found a bug or if you have a feature request, please [file an issue](#) on Github.

Example

Assume that you have the following contract in `Source.sol`:

```
pragma solidity >=0.6.0 <0.6.4;
// This will not compile after 0.7.0
// SPDX-License-Identifier: GPL-3.0
contract C {
    // FIXME: remove constructor visibility and make the contract abstract
    constructor() internal {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME: change now to block.timestamp
        time = now;
    }
}

contract E {
    D d;

    // FIXME: remove constructor visibility
    constructor() public {}

    function g() public {
        // FIXME: change .value(5) => {value: 5}
        d.f.value(5)();
    }
}
```

Required Changes

The above contract will not compile starting from 0.7.0. To bring the contract up to date with the current Solidity version, the following upgrade modules have to be executed: `constructor-visibility`, `now` and `dotsyntax`. Please read the documentation on [available modules](#) for further details.

Running the Upgrade

It is recommended to explicitly specify the upgrade modules by using `--modules` argument.

```
solidity-upgrade --modules constructor-visibility,now,dotsyntax Source.sol
```

The command above applies all changes as shown below. Please review them carefully (the pragmas will have to be updated manually.)

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
abstract contract C {
    // FIXME: remove constructor visibility and make the contract abstract
    constructor() {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME: change now to block.timestamp
        time = block.timestamp;
    }
}

contract E {
    D d;

    // FIXME: remove constructor visibility
    constructor() {}

    function g() public {
        // FIXME: change .value(5) => {value: 5}
        d.f{value: 5}();
    }
}
```

3.13 Analysing the Compiler Output

It is often useful to look at the assembly code generated by the compiler. The generated binary, i.e., the output of `solc --bin contract.sol`, is generally difficult to read. It is recommended to use the flag `--asm` to analyse the assembly output. Even for large contracts, looking at a visual diff of the assembly before and after a change is often very enlightening.

Consider the following contract (named, say `contract.sol`):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

The following would be the output of `solc --asm contract.sol`

```
===== contract.sol:C =====
EVM assembly:
/* "contract.sol":0:86  contract C {... */
mstore(0x40, 0x80)
callvalue
dup1
iszero
tag_1
jumpi
0x00
dup1
revert
tag_1:
pop
dataSize(sub_0)
dup1
dataOffset(sub_0)
0x00
codecopy
0x00
return
stop

sub_0: assembly {
/* "contract.sol":0:86  contract C {... */
mstore(0x40, 0x80)
callvalue
dup1
iszero
tag_1
jumpi
0x00
dup1
revert
```

(continues on next page)

(continued from previous page)

```

tag_1:
    pop
    jumpi(tag_2, lt(calldatasize, 0x04))
    shr(0xe0, calldataload(0x00))
    dup1
    0x901717d1
    eq
    tag_3
    jumpi
tag_2:
    0x00
    dup1
    revert
    /* "contract.sol":17:84  function one() public pure returns (uint) {... */
tag_3:
    tag_4
    tag_5
    jump // in
tag_4:
    mload(0x40)
    tag_6
    swap2
    swap1
    tag_7
    jump // in
tag_6:
    mload(0x40)
    dup1
    swap2
    sub
    swap1
    return
tag_5:
    /* "contract.sol":53:57  uint */
    0x00
    /* "contract.sol":76:77  1 */
    0x01
    /* "contract.sol":69:77  return 1 */
    swap1
    pop
    /* "contract.sol":17:84  function one() public pure returns (uint) {... */
    swap1
    jump // out
    /* "#utility.yul":7:125 */
tag_10:
    /* "#utility.yul":94:118 */
    tag_12
    /* "#utility.yul":112:117 */
    dup2
    /* "#utility.yul":94:118 */
    tag_13
    jump // in

```

(continues on next page)

(continued from previous page)

```

tag_12:
    /* "#utility.yul":89:92 */
    dup3
    /* "#utility.yul":82:119 */
    mstore
    /* "#utility.yul":72:125 */
    pop
    pop
    jump // out
    /* "#utility.yul":131:353 */
tag_7:
    0x00
    /* "#utility.yul":262:264 */
    0x20
    /* "#utility.yul":251:260 */
    dup3
    /* "#utility.yul":247:265 */
    add
    /* "#utility.yul":239:265 */
    swap1
    pop
    /* "#utility.yul":275:346 */
tag_15:
    /* "#utility.yul":343:344 */
    0x00
    /* "#utility.yul":332:341 */
    dup4
    /* "#utility.yul":328:345 */
    add
    /* "#utility.yul":319:325 */
    dup5
    /* "#utility.yul":275:346 */
tag_10:
    jump // in
tag_15:
    /* "#utility.yul":229:353 */
    swap3
    swap2
    pop
    pop
    jump // out
    /* "#utility.yul":359:436 */
tag_13:
    0x00
    /* "#utility.yul":425:430 */
    dup2
    /* "#utility.yul":414:430 */
    swap1
    pop
    /* "#utility.yul":404:436 */
    swap2
    swap1

```

(continues on next page)

(continued from previous page)

```

pop
jump // out

auxdata:_
0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}

```

Alternatively, the above output can also be obtained from [Remix](#), under the option “Compilation Details” after compiling a contract.

Notice that the `asm` output starts with the creation / constructor code. The deploy code is provided as part of the `sub` object (in the above example, it is part of the sub-object `sub_0`). The `auxdata` field corresponds to the contract *metadata*. The comments in the assembly output point to the source location. Note that `#utility.yul` is an internally generated file of utility functions that can be obtained using the flags `--combined-json generated-sources, generated-sources-runtime`.

Similarly, the optimized assembly can be obtained with the command: `solc --optimize --asm contract.sol`. Often times, it is interesting to see if two different sources in Solidity result in the same optimized code. For example, to see if the expressions `(a * b) / c`, `a * b / c` generates the same bytecode. This can be easily done by taking a `diff` of the corresponding assembly output, after potentially stripping comments that reference the source locations.

Note: The `--asm` output is not designed to be machine readable. Therefore, there may be breaking changes on the output between minor versions of `solc`.

3.14 Solidity IR-based Codegen Changes

Solidity can generate EVM bytecode in two different ways: Either directly from Solidity to EVM opcodes (“old codegen”) or through an intermediate representation (“IR”) in Yul (“new codegen” or “IR-based codegen”).

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.

You can enable it on the command line using `--via-ir` or with the option `{"viaIR": true}` in `standard.json` and we encourage everyone to try it out!

For several reasons, there are tiny semantic differences between the old and the IR-based code generator, mostly in areas where we would not expect people to rely on this behaviour anyway. This section highlights the main differences between the old and the IR-based codegen.

3.14.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- The order of state variable initialization has changed in case of inheritance.

The order used to be:

- All state variables are zero-initialized at the beginning.
- Evaluate base constructor arguments from most derived to most base contract.
- Initialize all state variables in the whole inheritance hierarchy from most base to most derived.
- Run the constructor, if present, for all contracts in the linearized hierarchy from most base to most derived.

New order:

- All state variables are zero-initialized at the beginning.
- Evaluate base constructor arguments from most derived to most base contract.
- For every contract in order from most base to most derived in the linearized hierarchy:
 1. Initialize state variables.
 2. Run the constructor (if present).

This causes differences in contracts where the initial value of a state variable relies on the result of the constructor in another contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}
contract B is A {
    uint public y = f();
}
```

Previously, `y` would be set to 0. This is due to the fact that we would first initialize state variables: First, `x` is set to 0, and when initializing `y`, `f()` would return 0 causing `y` to be 0 as well. With the new rules, `y` will be set to 42. We first initialize `x` to 0, then call `A`'s constructor which sets `x` to 42. Finally, when initializing `y`, `f()` returns 42 causing `y` to be 42.

- When storage structs are deleted, every storage slot that contains a member of the struct is set to zero entirely. Formerly, padding space was left untouched. Consequently, if the padding space within a struct is used to store data (e.g. in the context of a contract upgrade), you have to be aware that `delete` will now also clear the added member (while it wouldn't have been cleared in the past).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s occupies only first 16 bytes of the 32 bytes slot
        // delete will write zero to the full slot
    }
}
```

We have the same behavior for implicit delete, for example when array of structs is shortened.

- Function modifiers are implemented in a slightly different way regarding function parameters and return variables. This especially has an effect if the placeholder `_;` is evaluated multiple times in a modifier. In the old code generator, each function parameter and return variable has a fixed slot on the stack. If the function is run multiple times because `_;` is used multiple times or used in a loop, then a change to the function parameter's or return variable's value is visible in the next execution of the function. The new code generator implements modifiers using actual functions and passes function parameters on. This means that multiple evaluations of a function's body will get the same values for the parameters, and the effect on return variables is that they are reset to their default (zero) value for each execution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint a) public pure mod() returns (uint r) {
        r = a++;
    }
    modifier mod() { _; _; }
}
```

If you execute `f(0)` in the old code generator, it will return 2, while it will return 1 when using the new code generator.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract C {
    bool active = true;
    modifier mod()
    {
        _;
        active = false;
        _;
    }
    function foo() external mod() returns (uint ret)
    {
        if (active)
            ret = 1; // Same as ``return 1``
    }
}
```

The function `C.foo()` returns the following values:

- Old code generator: 1 as the return variable is initialized to `0` only once before the first `_;` evaluation and then overwritten by the `return 1;`. It is not initialized again for the second `_;` evaluation and `foo()` does not explicitly assign it either (due to `active == false`), thus it keeps its first value.
- New code generator: `0` as all parameters, including return parameters, will be re-initialized before each `_;` evaluation.
- For the old code generator, the evaluation order of expressions is unspecified. For the new code generator, we try to evaluate in source order (left to right), but do not guarantee it. This can lead to semantic differences.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function preincr_u8(uint8 a) public pure returns (uint8) {
        return ++a + a;
    }
}
```

The function `preincr_u8(1)` returns the following values:

- Old code generator: $3 (1 + 2)$ but the return value is unspecified in general
- New code generator: $4 (2 + 2)$ but the return value is not guaranteed

On the other hand, function argument expressions are evaluated in the same order by both code generators with the exception of the global functions `addmod` and `mulmod`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 a, uint8 b) public pure returns (uint8) {
        return a + b;
    }
    function g(uint8 a, uint8 b) public pure returns (uint8) {
        return add(++a + ++b, a + b);
    }
}
```

The function `g(1, 2)` returns the following values:

- Old code generator: $10 (\text{add}(2 + 3, 2 + 3))$ but the return value is unspecified in general
- New code generator: 10 but the return value is not guaranteed

The arguments to the global functions `addmod` and `mulmod` are evaluated right-to-left by the old code generator and left-to-right by the new code generator. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // Old code gen: add/mulmod(5, 4, 3)
        // New code gen: add/mulmod(4, 5, 5)
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

The function `f()` returns the following values:

- Old code generator: `aMod = 0` and `mMod = 2`
 - New code generator: `aMod = 4` and `mMod = 0`
- The new code generator imposes a hard limit of `type(uint64).max (0xffffffffffff)` for the free memory pointer. Allocations that would increase its value beyond this limit revert. The old code generator does not have this limit.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // allocation size: 576460752303423481
        // assumes freeMemPtr points to 0x80 initially
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -
→31) / 32;
        // freeMemPtr overflows UINT64_MAX
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

The function `f()` behaves as follows:

- Old code generator: runs out of gas while zeroing the array contents after the large memory allocation
- New code generator: reverts due to free memory pointer overflow (does not run out of gas)

3.14.2 Internals

Internal function pointers

The old code generator uses code offsets or tags for values of internal function pointers. This is especially complicated since these offsets are different at construction time and after deployment and the values can cross this border via storage. Because of that, both offsets are encoded at construction time into the same value (into different bytes).

In the new code generator, function pointers use internal IDs that are allocated in sequence. Since calls via jumps are not possible, calls through function pointers always have to use an internal dispatch function that uses the `switch` statement to select the right function.

The ID `0` is reserved for uninitialized function pointers which then cause a panic in the dispatch function when called.

In the old code generator, internal function pointers are initialized with a special function that always causes a panic. This causes a storage write at construction time for internal function pointers in storage.

Cleanup

The old code generator only performs cleanup before an operation whose result could be affected by the values of the dirty bits. The new code generator performs cleanup after any operation that can result in dirty bits. The hope is that the optimizer will be powerful enough to eliminate redundant cleanup operations.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 a) public pure returns (uint r1, uint r2)
    {
        a = ⊻a;
        assembly {
            r1 := a
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    r2 = a;
}
}

```

The function `f(1)` returns the following values:

- Old code generator: (fffffffffffffffffffffffffffffffffffffe, 00fe)
- New code generator: (00fe, 00fe)

Note that, unlike the new code generator, the old code generator does not perform a cleanup after the bit-not assignment (`a = ~a`). This results in different values being assigned (within the inline assembly block) to return value `r1` between the old and new code generators. However, both code generators perform a cleanup before the new value of `a` is assigned to `r2`.

3.15 Layout of State Variables in Storage

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings (see below), data is stored contiguously item after item starting with the first state variable, which is stored in slot `0`. For each variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Value types use only as many bytes as are necessary to store them.
- If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot.
- Structs and array data always start a new slot and their items are packed tightly according to these rules.
- Items following struct or array data always start a new storage slot.

For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

The elements of structs and arrays are stored after each other, just as if they were given as individual values.

Warning: When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. If you are not reading or writing all the values in a slot at the same time, this can have the opposite effect, though: When one value is written to a multi-value storage slot, the storage slot has to be read first and then combined with the new value such that other data in the same slot is not destroyed.

When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and struct members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

Note: The layout of state variables in storage is considered to be part of the external interface of Solidity due to the fact that storage pointers can be passed to libraries. This means that any change to the rules outlined in this section is considered a breaking change of the language and due to its critical nature should be considered very carefully before being executed. In the event of such a breaking change, we would want to release a compatibility mode in which the compiler would generate bytecode supporting the old layout.

3.15.1 Mappings and Dynamic Arrays

Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored “in between” the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the [rules above](#) and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

Assume the storage location of the mapping or array ends up being a slot `p` after applying [the storage layout rules](#). For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see [below](#)). For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations.

Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively. The location of element `x[i][j]`, where the type of `x` is `uint24[][]`, is computed as follows (again, assuming `x` itself is stored at slot `p`): The slot is `keccak256(keccak256(p) + i + floor(j / floor(256 / 24)))` and the element can be obtained from the slot data `v` using `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

The value corresponding to a mapping key `k` is located at `keccak256(h(k) . p)` where `.` is concatenation and `h` is a function that is applied to the key depending on its type:

- for value types, `h` pads the value to 32 bytes in the same way as when storing the value in memory.
- for strings and byte arrays, `h(k)` is just the unpadded data.

If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

As an example, consider the following contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

Let us compute the storage location of `data[4][9].c`. The position of the mapping itself is 1 (the variable `x` with 32 bytes precedes it). This means `data[4]` is stored at `keccak256(uint256(4) . uint256(1))`. The type of `data[4]` is again a mapping and the data for `data[4][9]` starts at slot `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. The slot offset of the member `c` inside the struct `S` is 1 because `a` and `b` are packed in a single slot. This means the slot for `data[4][9].c` is `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`. The type of the value is `uint256`, so it uses a single slot.

bytes and string

`bytes` and `string` are encoded identically. In general, the encoding is similar to `bytes1[]`, in the sense that there is a slot for the array itself and a data area that is computed using a `keccak256` hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

In particular: if the data is at most 31 bytes long, the elements are stored in the higher-order bytes (left aligned) and the lowest-order byte stores the value `length * 2`. For byte arrays that store data which is 32 or more bytes long, the main slot `p` stores `length * 2 + 1` and the data is stored as usual in `keccak256(p)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set: short (not set) and long (set).

Note: Handling invalidly encoded slots is currently not supported but may be added in the future. If you are compiling via IR, reading an invalidly encoded slot results in a `Panic(0x22)` error.

3.15.2 JSON Output

The storage layout of a contract can be requested via the [standard JSON interface](#). The output is a JSON object containing two keys, `storage` and `types`. The `storage` object is an array where each element has the following form:

```
{
  "astId": 2,
  "contract": "fileA:A",
  "label": "x",
  "offset": 0,
  "slot": "0",
  "type": "t_uint256"
}
```

The example above is the storage layout of `contract A { uint x; }` from source unit `fileA` and

- `astId` is the id of the AST node of the state variable's declaration
- `contract` is the name of the contract including its path as prefix
- `label` is the name of the state variable
- `offset` is the offset in bytes within the storage slot according to the encoding
- `slot` is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- `type` is an identifier used as key to the variable's type information (described in the following)

The given `type`, in this case `t_uint256` represents an element in `types`, which has the form:

```
{
  "encoding": "inplace",
```

(continues on next page)

(continued from previous page)

```

"label": "uint256",
"numberOfBytes": "32",
}

```

where

- encoding how the data is encoded in storage, where the possible values are:
 - `inplace`: data is laid out contiguously in storage (see [above](#)).
 - `mapping`: Keccak-256 hash-based method (see [above](#)).
 - `dynamic_array`: Keccak-256 hash-based method (see [above](#)).
 - `bytes`: single slot or Keccak-256 hash-based depending on the data size (see [above](#)).
- `label` is the canonical type name.
- `numberOfBytes` is the number of used bytes (as a decimal string). Note that if `numberOfBytes > 32` this means that more than one slot is used.

Some types have extra information besides the four above. Mappings contain its key and value types (again referencing an entry in this mapping of types), arrays have its base type, and structs list their members in the same format as the top-level storage (see [above](#)).

Note: The JSON output format of a contract's storage layout is still considered experimental and is subject to change in non-breaking releases of Solidity.

The following example shows a contract and its storage layout, containing value and reference types, types that are encoded packed, and nested types.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping (uint => mapping (address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}

```

```
{
  "storage": [
    {
      "astId": 15,

```

(continues on next page)

(continued from previous page)

```

"contract": "fileA:A",
"label": "x",
"offset": 0,
"slot": "0",
"type": "t_uint256"
},
{
"astId": 17,
"contract": "fileA:A",
"label": "y",
"offset": 0,
"slot": "1",
"type": "t_uint256"
},
{
"astId": 20,
"contract": "fileA:A",
"label": "s",
"offset": 0,
"slot": "2",
"type": "t_struct(S)13_storage"
},
{
"astId": 22,
"contract": "fileA:A",
"label": "addr",
"offset": 0,
"slot": "6",
"type": "t_address"
},
{
"astId": 28,
"contract": "fileA:A",
"label": "map",
"offset": 0,
"slot": "7",
"type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
},
{
"astId": 31,
"contract": "fileA:A",
"label": "array",
"offset": 0,
"slot": "8",
"type": "t_array(t_uint256)dyn_storage"
},
{
"astId": 33,
"contract": "fileA:A",
"label": "s1",
"offset": 0,
"slot": "9",

```

(continues on next page)

(continued from previous page)

```

    "type": "t_string_storage"
},
{
  "astId": 35,
  "contract": "fileA:A",
  "label": "b1",
  "offset": 0,
  "slot": "10",
  "type": "t_bytes_storage"
}
],
"types": {
  "t_address": {
    "encoding": "inplace",
    "label": "address",
    "numberOfBytes": "20"
  },
  "t_array(t_uint256)2_storage": {
    "base": "t_uint256",
    "encoding": "inplace",
    "label": "uint256[2]",
    "numberOfBytes": "64"
  },
  "t_array(t_uint256)dyn_storage": {
    "base": "t_uint256",
    "encoding": "dynamic_array",
    "label": "uint256[]",
    "numberOfBytes": "32"
  },
  "t_bool": {
    "encoding": "inplace",
    "label": "bool",
    "numberOfBytes": "1"
  },
  "t_bytes_storage": {
    "encoding": "bytes",
    "label": "bytes",
    "numberOfBytes": "32"
  },
  "t_mapping(t_address,t_bool)": {
    "encoding": "mapping",
    "key": "t_address",
    "label": "mapping(address => bool)",
    "numberOfBytes": "32",
    "value": "t_bool"
  },
  "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
    "encoding": "mapping",
    "key": "t_uint256",
    "label": "mapping(uint256 => mapping(address => bool))",
    "numberOfBytes": "32",
    "value": "t_mapping(t_address,t_bool)"
  }
}

```

(continues on next page)

(continued from previous page)

```

},
"t_string_storage": {
  "encoding": "bytes",
  "label": "string",
  "numberOfBytes": "32"
},
"t_struct(S)13_storage": {
  "encoding": "inplace",
  "label": "struct A.S",
  "members": [
    {
      "astId": 3,
      "contract": "fileA:A",
      "label": "a",
      "offset": 0,
      "slot": "0",
      "type": "t_uint128"
    },
    {
      "astId": 5,
      "contract": "fileA:A",
      "label": "b",
      "offset": 16,
      "slot": "0",
      "type": "t_uint128"
    },
    {
      "astId": 9,
      "contract": "fileA:A",
      "label": "staticArray",
      "offset": 0,
      "slot": "1",
      "type": "t_array(t_uint256)2_storage"
    },
    {
      "astId": 12,
      "contract": "fileA:A",
      "label": "dynArray",
      "offset": 0,
      "slot": "3",
      "type": "t_array(t_uint256)dyn_storage"
    }
  ],
  "numberOfBytes": "128"
},
"t_uint128": {
  "encoding": "inplace",
  "label": "uint128",
  "numberOfBytes": "16"
},
"t_uint256": {
  "encoding": "inplace",

```

(continues on next page)

(continued from previous page)

```

    "label": "uint256",
    "numberOfBytes": "32"
}
}
}

```

3.16 Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

- **0x00 - 0x3f** (64 bytes): scratch space for hashing methods
- **0x40 - 0x5f** (32 bytes): currently allocated memory size (aka. free memory pointer)
- **0x60 - 0x7f** (32 bytes): zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to **0x80** initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Warning: There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one should not expect the free memory to point to zeroed out memory.

While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have unexpected results.

3.16.1 Differences to Layout in Storage

As described above the layout in memory is different from the layout in *storage*. Below there are some examples.

Example for Difference in Arrays

The following array occupies 32 bytes (1 slot) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
uint8[4] a;
```

Example for Difference in Struct Layout

The following struct occupies 96 bytes (3 slots of 32 bytes) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

3.17 Layout of Call Data

The input data for a function call is assumed to be in the format defined by the [ABI specification](#). Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

3.18 Cleaning Up Variables

When a value is shorter than 256 bit, in some cases the remaining bits must be cleaned. The Solidity compiler is designed to clean such remaining bits before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

Note that access via inline assembly is not considered such an operation: If you use inline assembly to access Solidity variables shorter than 256 bits, the compiler does not guarantee that the value is properly cleaned up.

Moreover, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

Different types have different rules for cleaning up invalid values:

Type	Valid Values	Invalid Values Mean
enum of n members	0 until n - 1	exception
bool	0 or 1	1
signed integers	sign-extended word	currently silently wraps; in the future exceptions will be thrown
unsigned integers	higher bits zeroed	currently silently wraps; in the future exceptions will be thrown

3.19 Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling. This mapping also contains other information, like the jump type and the modifier depth (see below).

Both kinds of source mappings use integer identifiers to refer to source files. The identifier of a source file is stored in `output['sources'][sourceName]['id']` where `output` is the output of the standard-json compiler interface parsed as JSON. For some utility routines, the compiler generates “internal” source files that are not part of the original input but are referenced from the source mappings. These source files together with their identifiers can be obtained via `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']`.

Note: In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of -1. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

The source mappings inside the AST use the following notation:

`s:l:f`

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated: It is a list of `s:l:f:j:m` separated by ;. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above. `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop. The last field, `m`, is an integer that denotes the “modifier depth”. This depth is increased whenever the placeholder statement `(_)` is entered in a modifier and decreased when it is left again. This allows debuggers to track tricky cases like the same modifier being used twice or multiple placeholder statements being used in a single modifier.

In order to compress these source mappings especially for bytecode, the following rules are used:

- If a field is empty, the value of the preceding element is used.
- If a : is missing, all following fields are considered empty.

This means the following source mappings represent the same information:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

Important to note is that when the `verbatim` builtin is used, the source mappings will be invalid: The builtin is considered a single instruction instead of potentially multiple.

3.20 The Optimizer

The Solidity compiler uses two different optimizer modules: The “old” optimizer that operates at the opcode level and the “new” optimizer that operates on Yul IR code.

The opcode-based optimizer applies a set of simplification rules to opcodes. It also combines equal code sets and removes unused code.

The Yul-based optimizer is much more powerful, because it can work across function calls. For example, arbitrary jumps are not possible in Yul, so it is possible to compute the side-effects of each function. Consider two function calls, where the first does not modify storage and the second does modify storage. If their arguments and return values do not depend on each other, we can reorder the function calls. Similarly, if a function is side-effect free and its result is multiplied by zero, you can remove the function call completely.

Currently, the parameter `--optimize` activates the opcode-based optimizer for the generated bytecode and the Yul optimizer for the Yul code generated internally, for example for ABI coder v2. One can use `solc --ir-optimized --optimize` to produce an optimized Yul IR for a Solidity source. Similarly, one can use `solc --strict-assembly --optimize` for a stand-alone Yul mode.

You can find more details on both optimizer modules and their optimization steps below.

3.20.1 Benefits of Optimizing Solidity Code

Overall, the optimizer tries to simplify complicated expressions, which reduces both code size and execution cost, i.e., it can reduce gas needed for contract deployment as well as for external calls made to the contract. It also specializes or inlines functions. Especially function inlining is an operation that can cause much bigger code, but it is often done because it results in opportunities for more simplifications.

3.20.2 Differences between Optimized and Non-Optimized Code

Generally, the most visible difference is that constant expressions are evaluated at compile time. When it comes to the ASM output, one can also notice a reduction of equivalent or duplicate code blocks (compare the output of the flags `--asm` and `--asm --optimize`). However, when it comes to the Yul/intermediate-representation, there can be significant differences, for example, functions may be inlined, combined, or rewritten to eliminate redundancies, etc. (compare the output between the flags `--ir` and `--optimize --ir-optimized`).

3.20.3 Optimizer Parameter Runs

The number of runs (`--optimize-runs`) specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract. This means it is a trade-off parameter between code size (deploy cost) and code execution cost (cost after deployment). A “runs” parameter of “1” will produce short but expensive code. In contrast, a larger “runs” parameter will produce longer but more gas efficient code. The maximum value of the parameter is $2^{32}-1$.

Note: A common misconception is that this parameter specifies the number of iterations of the optimizer. This is not true: The optimizer will always run as many times as it can still improve the code.

3.20.4 Opcode-Based Optimizer Module

The opcode-based optimizer module operates on assembly code. It splits the sequence of instructions into basic blocks at JUMPs and JUMPDESTs. Inside these blocks, the optimizer analyzes the instructions and records every modification to the stack, memory, or storage as an expression which consists of an instruction and a list of arguments which are pointers to other expressions.

Additionally, the opcode-based optimizer uses a component called “CommonSubexpressionEliminator” that, amongst other tasks, finds expressions that are always equal (on every input) and combines them into an expression class. It first tries to find each new expression in a list of already known expressions. If no such matches are found, it simplifies the expression according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is a recursive process, we can also apply the latter rule if the second factor is a more complex expression which we know always evaluates to one.

Certain optimizer steps symbolically track the storage and memory locations. For example, this information is used to compute Keccak-256 hashes that can be evaluated during compile time. Consider the sequence:

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

or the equivalent Yul

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

In this case, the optimizer tracks the value at a memory location `calldataload(0)` and then realizes that the Keccak-256 hash can be evaluated at compile time. This only works if there is no other instruction that modifies memory between the `mstore` and `keccak256`. So if there is an instruction that writes to memory (or storage), then we need to erase the knowledge of the current memory (or storage). There is, however, an exception to this erasing, when we can easily see that the instruction doesn't write to a certain location.

For example,

```
let x := calldataload(0)
mstore(x, 100)
// Current knowledge memory location x -> 100
let y := add(x, 32)
// Does not clear the knowledge that x -> 100, since y does not write to [x, x + 32)
mstore(y, 200)
// This Keccak-256 can now be evaluated
let value := keccak256(x, 32)
```

Therefore, modifications to storage and memory locations, of say location 1, must erase knowledge about storage or memory locations which may be equal to 1. More specifically, for storage, the optimizer has to erase all knowledge of symbolic locations, that may be equal to 1 and for memory, the optimizer has to erase all knowledge of symbolic locations that may not be at least 32 bytes away. If `m` denotes an arbitrary location, then this decision on erasure is done by computing the value `sub(1, m)`. For storage, if this value evaluates to a literal that is non-zero, then the knowledge about `m` will be kept. For memory, if the value evaluates to a literal that is between 32 and $2^{**}256 - 32$, then the knowledge about `m` will be kept. In all other cases, the knowledge about `m` will be erased.

After this process, we know which expressions have to be on the stack at the end, and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s).

If we know the targets of all JUMP and JUMPI instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown JUMP. If the opcode-based optimizer module finds a JUMPI whose condition evaluates to a constant, it transforms it to an unconditional jump.

As the last step, the code in each block is re-generated. The optimizer creates a dependency graph from the expressions on the stack at the end of the block, and it drops every operation that is not part of this graph. It generates code that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed). Finally, it generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a JUMPI and during the analysis, the condition evaluates to a constant, the JUMPI is replaced based on the value of the constant. Thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // this condition is never true
    return 2;
else
    return 1;
```

simplifies to this:

```
data[7] = 9;
return 1;
```

Simple Inlining

Since Solidity version 0.8.2, there is another optimizer step that replaces certain jumps to blocks containing “simple” instructions ending with a “jump” by a copy of these instructions. This corresponds to inlining of simple, small Solidity or Yul functions. In particular, the sequence PUSHTAG(tag) JUMP may be replaced, whenever the JUMP is marked as jump “into” a function and behind tag there is a basic block (as described above for the “CommonSubexpressionEliminator”) that ends in another JUMP which is marked as a jump “out of” a function.

In particular, consider the following prototypical example of assembly generated for a call to an internal Solidity function:

```
tag_return
tag_f
jump      // in
tag_return:
...opcodes after call to f...

tag_f:
...body of function f...
jump      // out
```

As long as the body of the function is a continuous basic block, the “Inliner” can replace tag_f jump by the block at tag_f resulting in:

```
tag_return
...body of function f...
jump
tag_return:
...opcodes after call to f...

tag_f:
...body of function f...
jump // out
```

Now ideally, the other optimizer steps described above will result in the return tag push being moved towards the remaining jump resulting in:

```
...body of function f...
tag_return
jump
tag_return:
...opcodes after call to f...

tag_f:
...body of function f...
jump // out
```

In this situation the “PeepholeOptimizer” will remove the return jump. Ideally, all of this can be done for all references to `tag_f` leaving it unused, s.t. it can be removed, yielding:

```
...body of function f...
...opcodes after call to f...
```

So the call to function `f` is inlined and the original definition of `f` can be removed.

Inlining like this is attempted, whenever a heuristics suggests that inlining is cheaper over the lifetime of a contract than not inlining. This heuristics depends on the size of the function body, the number of other references to its tag (approximating the number of calls to the function) and the expected number of executions of the contract (the global optimizer parameter “runs”).

3.20.5 Yul-Based Optimizer Module

The Yul-based optimizer consists of several stages and components that all transform the AST in a semantically equivalent way. The goal is to end up either with code that is shorter or at least only marginally longer but will allow further optimization steps.

Warning: Since the optimizer is under heavy development, the information here might be outdated. If you rely on a certain functionality, please reach out to the team directly.

The optimizer currently follows a purely greedy strategy and does not do any backtracking.

All components of the Yul-based optimizer module are explained below. The following transformation steps are the main components:

- SSA Transform
- Common Subexpression Eliminator

- Expression Simplifier
- Redundant Assign Eliminator
- Full Inliner

Optimizer Steps

This is a list of all steps the Yul-based optimizer sorted alphabetically. You can find more information on the individual steps and their sequence below.

- *BlockFlattener*.
- *CircularReferencesPruner*.
- *CommonSubexpressionEliminator*.
- *ConditionalSimplifier*.
- *ConditionalUnsimplifier*.
- *ControlFlowSimplifier*.
- *DeadCodeEliminator*.
- *EqualStoreEliminator*.
- *EquivalentFunctionCombiner*.
- *ExpressionJoiner*.
- *Expression Simplifier*.
- *ExpressionSplitter*.
- *ForLoopConditionIntoBody*.
- *ForLoopConditionOutOfBody*.
- *ForLoopInitRewriter*.
- *ExpressionInliner*.
- *FullInliner*.
- *FunctionGrouper*.
- *FunctionHoister*.
- *FunctionSpecializer*.
- *LiteralRematerialiser*.
- *LoadResolver*.
- *LoopInvariantCodeMotion*.
- *RedundantAssignEliminator*.
- *ReasoningBasedSimplifier*.
- *Rematerialiser*.
- *SSAReverser*.
- *SSATransform*.
- *StructuralSimplifier*.

- *UnusedFunctionParameterPruner*.
- *UnusedPruner*.
- *VarDeclInitializer*.

Selecting Optimizations

By default the optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

The sequence inside [...] will be applied multiple times in a loop until the Yul code remains unchanged or until the maximum number of rounds (currently 12) has been reached.

Available abbreviations are listed in the [Yul optimizer docs](#).

Preprocessing

The preprocessing components perform transformations to get the program into a certain normal form that is easier to work with. This normal form is kept during the rest of the optimization process.

Disambiguator

The disambiguator takes an AST and returns a fresh copy where all identifiers have unique names in the input AST. This is a prerequisite for all other optimizer stages. One of the benefits is that identifier lookup does not need to take scopes into account which simplifies the analysis needed for other steps.

All subsequent stages have the property that all names stay unique. This means if a new identifier needs to be introduced, a new unique name is generated.

FunctionHoister

The function hoister moves all function definitions to the end of the topmost block. This is a semantically equivalent transformation as long as it is performed after the disambiguation stage. The reason is that moving a definition to a higher-level block cannot decrease its visibility and it is impossible to reference variables defined in a different function.

The benefit of this stage is that function definitions can be looked up more easily and functions can be optimized in isolation without having to traverse the AST completely.

FunctionGrouper

The function grouper has to be applied after the disambiguator and the function hoister. Its effect is that all topmost elements that are not function definitions are moved into a single block which is the first statement of the root block.

After this step, a program has the following normal form:

```
{ I F... }
```

Where I is a (potentially empty) block that does not contain any function definitions (not even recursively) and F is a list of function definitions such that no function contains a function definition.

The benefit of this stage is that we always know where the list of function begins.

ForLoopConditionIntoBody

This transformation moves the loop-iteration condition of a for-loop into loop body. We need this transformation because [ExpressionSplitter](#) will not apply to iteration condition expressions (the C in the following example).

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
for { Init... } 1 { Post... } {
    if iszero(C) { break }
    Body...
}
```

This transformation can also be useful when paired with [LoopInvariantCodeMotion](#), since invariants in the loop-invariant conditions can then be taken outside the loop.

ForLoopInitRewriter

This transformation moves the initialization part of a for-loop to before the loop:

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
Init...
for {} C { Post... } {
    Body...
}
```

This eases the rest of the optimization process because we can ignore the complicated scoping rules of the for loop initialisation block.

VarDeclInitializer

This step rewrites variable declarations so that all of them are initialized. Declarations like `let x, y` are split into multiple declaration statements.

Only supports initializing with the zero literal for now.

Pseudo-SSA Transformation

The purpose of this components is to get the program into a longer form, so that other components can more easily work with it. The final representation will be similar to a static-single-assignment (SSA) form, with the difference that it does not make use of explicit “phi” functions which combines the values from different branches of control flow because such a feature does not exist in the Yul language. Instead, when control flow merges, if a variable is re-assigned in one of the branches, a new SSA variable is declared to hold its current value, so that the following expressions still only need to reference SSA variables.

An example transformation is the following:

```
{  
    let a := calldataload(0)  
    let b := calldataload(0x20)  
    if gt(a, 0) {  
        b := mul(b, 0x20)  
    }  
    a := add(a, 1)  
    sstore(a, add(b, 0x20))  
}
```

When all the following transformation steps are applied, the program will look as follows:

```
{  
    let _1 := 0  
    let a_9 := calldataload(_1)  
    let a := a_9  
    let _2 := 0x20  
    let b_10 := calldataload(_2)  
    let b := b_10  
    let _3 := 0  
    let _4 := gt(a_9, _3)  
    if _4  
    {  
        let _5 := 0x20  
        let b_11 := mul(b_10, _5)  
        b := b_11  
    }  
    let b_12 := b  
    let _6 := 1  
    let a_13 := add(a_9, _6)  
    let _7 := 0x20  
    let _8 := add(b_12, _7)  
    sstore(a_13, _8)  
}
```

Note that the only variable that is re-assigned in this snippet is b. This re-assignment cannot be avoided because b has different values depending on the control flow. All other variables never change their value once they are defined. The advantage of this property is that variables can be freely moved around and references to them can be exchanged by their initial value (and vice-versa), as long as these values are still valid in the new context.

Of course, the code here is far from being optimized. To the contrary, it is much longer. The hope is that this code will be easier to work with and furthermore, there are optimizer steps that undo these changes and make the code more compact again at the end.

ExpressionSplitter

The expression splitter turns expressions like `add(mload(0x123), mul(mload(0x456), 0x20))` into a sequence of declarations of unique variables that are assigned sub-expressions of that expression so that each function call has only variables as arguments.

The above would be transformed into

```
{
    let _1 := 0x20
    let _2 := 0x456
    let _3 := mload(_2)
    let _4 := mul(_3, _1)
    let _5 := 0x123
    let _6 := mload(_5)
    let z := add(_6, _4)
}
```

Note that this transformation does not change the order of opcodes or function calls.

It is not applied to loop iteration-condition, because the loop control flow does not allow this “outlining” of the inner expressions in all cases. We can sidestep this limitation by applying [ForLoopConditionIntoBody](#) to move the iteration condition into loop body.

The final program should be in a form such that (with the exception of loop conditions) function calls cannot appear nested inside expressions and all function call arguments have to be variables.

The benefits of this form are that it is much easier to re-order the sequence of opcodes and it is also easier to perform function call inlining. Furthermore, it is simpler to replace individual parts of expressions or re-organize the “expression tree”. The drawback is that such code is much harder to read for humans.

SSATransform

This stage tries to replace repeated assignments to existing variables by declarations of new variables as much as possible. The reassigned variables are still there, but all references to the reassigned variables are replaced by the newly declared variables.

Example:

```
{
    let a := 1
    mstore(a, 2)
    a := 3
}
```

is transformed to

```
{
    let a_1 := 1
    let a := a_1
    mstore(a_1, 2)
    let a_3 := 3
    a := a_3
}
```

Exact semantics:

For any variable `a` that is assigned to somewhere in the code (variables that are declared with value and never re-assigned are not modified) perform the following transforms:

- replace `let a := v` by `let a_i := v let a := a_i`
- replace `a := v` by `let a_i := v a := a_i` where `i` is a number such that `a_i` is yet unused.

Furthermore, always record the current value of `i` used for `a` and replace each reference to `a` by `a_i`. The current value mapping is cleared for a variable `a` at the end of each block in which it was assigned to and at the end of the for loop init block if it is assigned inside the for loop body or post block. If a variable's value is cleared according to the rule above and the variable is declared outside the block, a new SSA variable will be created at the location where control flow joins, this includes the beginning of loop post/body block and the location right after If/Switch/ForLoop/Block statement.

After this stage, the Redundant Assign Eliminator is recommended to remove the unnecessary intermediate assignments.

This stage provides best results if the Expression Splitter and the Common Subexpression Eliminator are run right before it, because then it does not generate excessive amounts of variables. On the other hand, the Common Subexpression Eliminator could be more efficient if run after the SSA transform.

RedundantAssignEliminator

The SSA transform always generates an assignment of the form `a := a_i`, even though these might be unnecessary in many cases, like the following example:

```
{
    let a := 1
    a := mload(a)
    a := sload(a)
    sstore(a, 1)
}
```

The SSA transform converts this snippet to the following:

```
{
    let a_1 := 1
    let a := a_1
    let a_2 := mload(a_1)
    a := a_2
    let a_3 := sload(a_2)
    a := a_3
    sstore(a_3, 1)
}
```

The Redundant Assign Eliminator removes all the three assignments to `a`, because the value of `a` is not used and thus turn this snippet into strict SSA form:

```
{
    let a_1 := 1
    let a_2 := mload(a_1)
    let a_3 := sload(a_2)
    sstore(a_3, 1)
}
```

Of course the intricate parts of determining whether an assignment is redundant or not are connected to joining control flow.

The component works as follows in detail:

The AST is traversed twice: in an information gathering step and in the actual removal step. During information gathering, we maintain a mapping from assignment statements to the three states “unused”, “undecided” and “used” which signifies whether the assigned value will be used later by a reference to the variable.

When an assignment is visited, it is added to the mapping in the “undecided” state (see remark about for loops below) and every other assignment to the same variable that is still in the “undecided” state is changed to “unused”. When a variable is referenced, the state of any assignment to that variable still in the “undecided” state is changed to “used”.

At points where control flow splits, a copy of the mapping is handed over to each branch. At points where control flow joins, the two mappings coming from the two branches are combined in the following way: Statements that are only in one mapping or have the same state are used unchanged. Conflicting values are resolved in the following way:

- “unused”, “undecided” -> “undecided”
- “unused”, “used” -> “used”
- “undecided”, “used” -> “used”

For for-loops, the condition, body and post-part are visited twice, taking the joining control-flow at the condition into account. In other words, we create three control flow paths: Zero runs of the loop, one run and two runs and then combine them at the end.

Simulating a third run or even more is unnecessary, which can be seen as follows:

A state of an assignment at the beginning of the iteration will deterministically result in a state of that assignment at the end of the iteration. Let this state mapping function be called f . The combination of the three different states unused, undecided and used as explained above is the `max` operation where `unused = 0`, `undecided = 1` and `used = 2`.

The proper way would be to compute

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

as state after the loop. Since f just has a range of three different values, iterating it has to reach a cycle after at most three iterations, and thus $f(f(f(s)))$ has to equal one of s , $f(s)$, or $f(f(s))$ and thus

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...).
```

In summary, running the loop at most twice is enough because there are only three different states.

For switch statements that have a “default”-case, there is no control-flow part that skips the switch.

When a variable goes out of scope, all statements still in the “undecided” state are changed to “unused”, unless the variable is the return parameter of a function - there, the state changes to “used”.

In the second traversal, all assignments that are in the “unused” state are removed.

This step is usually run right after the SSA transform to complete the generation of the pseudo-SSA.

Tools

Movability

Movability is a property of an expression. It roughly means that the expression is side-effect free and its evaluation only depends on the values of variables and the call-constant state of the environment. Most expressions are movable. The following parts make an expression non-movable:

- function calls (might be relaxed in the future if all statements in the function are movable)
- opcodes that (can) have side-effects (like `call` or `selfdestruct`)
- opcodes that read or write memory, storage or external state information
- opcodes that depend on the current PC, memory size or `returndata` size

DataflowAnalyzer

The Dataflow Analyzer is not an optimizer step itself but is used as a tool by other components. While traversing the AST, it tracks the current value of each variable, as long as that value is a movable expression. It records the variables that are part of the expression that is currently assigned to each other variable. Upon each assignment to a variable `a`, the current stored value of `a` is updated and all stored values of all variables `b` are cleared whenever `a` is part of the currently stored expression for `b`.

At control-flow joins, knowledge about variables is cleared if they have or would be assigned in any of the control-flow paths. For instance, upon entering a for loop, all variables are cleared that will be assigned during the body or the post block.

Expression-Scale Simplifications

These simplification passes change expressions and replace them by equivalent and hopefully simpler expressions.

CommonSubexpressionEliminator

This step uses the Dataflow Analyzer and replaces subexpressions that syntactically match the current value of a variable by a reference to that variable. This is an equivalence transform because such subexpressions have to be movable.

All subexpressions that are identifiers themselves are replaced by their current value if the value is an identifier.

The combination of the two rules above allow to compute a local value numbering, which means that if two variables have the same value, one of them will always be unused. The Unused Pruner or the Redundant Assign Eliminator will then be able to fully eliminate such variables.

This step is especially efficient if the expression splitter is run before. If the code is in pseudo-SSA form, the values of variables are available for a longer time and thus we have a higher chance of expressions to be replaceable.

The expression simplifier will be able to perform better replacements if the common subexpression eliminator was run right before it.

Expression Simplifier

The Expression Simplifier uses the Dataflow Analyzer and makes use of a list of equivalence transforms on expressions like $X + 0 \rightarrow X$ to simplify the code.

It tries to match patterns like $X + 0$ on each subexpression. During the matching procedure, it resolves variables to their currently assigned expressions to be able to match more deeply nested patterns even when the code is in pseudo-SSA form.

Some of the patterns like $X - X \rightarrow 0$ can only be applied as long as the expression X is movable, because otherwise it would remove its potential side-effects. Since variable references are always movable, even if their current value might not be, the Expression Simplifier is again more powerful in split or pseudo-SSA form.

LiteralRematerialiser

To be documented.

LoadResolver

Optimisation stage that replaces expressions of type `sload(x)` and `mload(x)` by the value currently stored in storage resp. memory, if known.

Works best if the code is in SSA form.

Prerequisite: Disambiguator, ForLoopInitRewriter.

ReasoningBasedSimplifier

This optimizer uses SMT solvers to check whether `if` conditions are constant.

- If `constraints AND condition` is UNSAT, the condition is never true and the whole body can be removed.
- If `constraints AND NOT condition` is UNSAT, the condition is always true and can be replaced by 1.

The simplifications above can only be applied if the condition is movable.

It is only effective on the EVM dialect, but safe to use on other dialects.

Prerequisite: Disambiguator, SSATransform.

Statement-Scale Simplifications

CircularReferencesPruner

This stage removes functions that call each other but are neither externally referenced nor referenced from the outermost context.

ConditionalSimplifier

The Conditional Simplifier inserts assignments to condition variables if the value can be determined from the control-flow.

Destroys SSA form.

Currently, this tool is very limited, mostly because we do not yet have support for boolean types. Since conditions only check for expressions being nonzero, we cannot assign a specific value.

Current features:

- switch cases: insert “<condition> := <caseLabel>”
- after if statement with terminating control-flow, insert “<condition> := 0”

Future features:

- allow replacements by “1”
- take termination of user-defined functions into account

Works best with SSA form and if dead code removal has run before.

Prerequisite: Disambiguator.

ConditionalUnsimplifier

Reverse of Conditional Simplifier.

ControlFlowSimplifier

Simplifies several control-flow structures:

- replace if with empty body with pop(condition)
- remove empty default switch case
- remove empty switch case if no default case exists
- replace switch with no cases with pop(expression)
- turn switch with single case into if
- replace switch with only default case with pop(expression) and body
- replace switch with const expr with matching case body
- replace `for` with terminating control flow and without other break/continue by `if`
- remove `leave` at the end of a function.

None of these operations depend on the data flow. The StructuralSimplifier performs similar tasks that do depend on data flow.

The ControlFlowSimplifier does record the presence or absence of `break` and `continue` statements during its traversal.

Prerequisite: Disambiguator, FunctionHoister, ForLoopInitRewriter. Important: Introduces EVM opcodes and thus can only be used on EVM code for now.

DeadCodeEliminator

This optimization stage removes unreachable code.

Unreachable code is any code within a block which is preceded by a leave, return, invalid, break, continue, selfdestruct or revert.

Function definitions are retained as they might be called by earlier code and thus are considered reachable.

Because variables declared in a for loop's init block have their scope extended to the loop body, we require ForLoopInitRewriter to run before this step.

Prerequisite: ForLoopInitRewriter, Function Hoister, Function Grouper

EqualStoreEliminator

This step removes `mstore(k, v)` and `sstore(k, v)` calls if there was a previous call to `mstore(k, v)` / `sstore(k, v)`, no other store in between and the values of k and v did not change.

This simple step is effective if run after the SSA transform and the Common Subexpression Eliminator, because SSA will make sure that the variables will not change and the Common Subexpression Eliminator re-uses exactly the same variable if the value is known to be the same.

Prerequisites: Disambiguator, ForLoopInitRewriter

UnusedPruner

This step removes the definitions of all functions that are never referenced.

It also removes the declaration of variables that are never referenced. If the declaration assigns a value that is not movable, the expression is retained, but its value is discarded.

All movable expression statements (expressions that are not assigned) are removed.

StructuralSimplifier

This is a general step that performs various kinds of simplifications on a structural level:

- replace if statement with empty body by `pop(condition)`
- replace if statement with true condition by its body
- remove if statement with false condition
- turn switch with single case into if
- replace switch with only default case by `pop(expression)` and body
- replace switch with literal expression by matching case body
- replace for loop with false condition by its initialization part

This component uses the Dataflow Analyzer.

BlockFlattener

This stage eliminates nested blocks by inserting the statement in the inner block at the appropriate place in the outer block. It depends on the FunctionGrouper and does not flatten the outermost block to keep the form produced by the FunctionGrouper.

```
{  
  {  
    let x := 2  
    {  
      let y := 3  
      mstore(x, y)  
    }  
  }  
}
```

is transformed to

```
{  
  {  
    let x := 2  
    let y := 3  
    mstore(x, y)  
  }  
}
```

As long as the code is disambiguated, this does not cause a problem because the scopes of variables can only grow.

LoopInvariantCodeMotion

This optimization moves movable SSA variable declarations outside the loop.

Only statements at the top level in a loop's body or post block are considered, i.e variable declarations inside conditional branches will not be moved out of the loop.

Requirements:

- The Disambiguator, ForLoopInitRewriter and FunctionHoister must be run upfront.
- Expression splitter and SSA transform should be run upfront to obtain better result.

Function-Level Optimizations

FunctionSpecializer

This step specializes the function with its literal arguments.

If a function, say, `function f(a, b) { sstore (a, b) }`, is called with literal arguments, for example, `f(x, 5)`, where `x` is an identifier, it could be specialized by creating a new function `f_1` that takes only one argument, i.e.,

```
function f_1(a_1) {  
  let b_1 := 5  
  sstore(a_1, b_1)  
}
```

Other optimization steps will be able to make more simplifications to the function. The optimization step is mainly useful for functions that would not be inlined.

Prerequisites: Disambiguator, FunctionHoister

LiteralRematerialiser is recommended as a prerequisite, even though it's not required for correctness.

UnusedFunctionParameterPruner

This step removes unused parameters in a function.

If a parameter is unused, like `c` and `y` in, `function f(a,b,c) -> x, y { x := div(a,b) }`, we remove the parameter and create a new “linking” function as follows:

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

and replace all references to `f` by `f2`. The inliner should be run afterwards to make sure that all references to `f2` are replaced by `f`.

Prerequisites: Disambiguator, FunctionHoister, LiteralRematerialiser.

The step LiteralRematerialiser is not required for correctness. It helps deal with cases such as: `function f(x) -> y { revert(y, y) }` where the literal `y` will be replaced by its value `0`, allowing us to rewrite the function.

EquivalentFunctionCombiner

If two functions are syntactically equivalent, while allowing variable renaming but not any re-ordering, then any reference to one of the functions is replaced by the other.

The actual removal of the function is performed by the Unused Pruner.

Function Inlining

ExpressionInliner

This component of the optimizer performs restricted function inlining by inlining functions that can be inlined inside functional expressions, i.e. functions that:

- return a single value.
- have a body like `r := <functional expression>`.
- neither reference themselves nor `r` in the right hand side.

Furthermore, for all parameters, all of the following need to be true:

- The argument is movable.
- The parameter is either referenced less than twice in the function body, or the argument is rather cheap (“cost” of at most 1, like a constant up to `0xff`).

Example: The function to be inlined has the form of `function f(...) -> r { r := E }` where `E` is an expression that does not reference `r` and all arguments in the function call are movable expressions.

The result of this inlining is always a single expression.

This component can only be used on sources with unique names.

FullInliner

The Full Inliner replaces certain calls of certain functions by the function's body. This is not very helpful in most cases, because it just increases the code size but does not have a benefit. Furthermore, code is usually very expensive and we would often rather have shorter code than more efficient code. In some cases, though, inlining a function can have positive effects on subsequent optimizer steps. This is the case if one of the function arguments is a constant, for example.

During inlining, a heuristic is used to tell if the function call should be inlined or not. The current heuristic does not inline into "large" functions unless the called function is tiny. Functions that are only used once are inlined, as well as medium-sized functions, while function calls with constant arguments allow slightly larger functions.

In the future, we may include a backtracking component that, instead of inlining a function right away, only specializes it, which means that a copy of the function is generated where a certain parameter is always replaced by a constant. After that, we can run the optimizer on this specialized function. If it results in heavy gains, the specialized function is kept, otherwise the original function is used instead.

Cleanup

The cleanup is performed at the end of the optimizer run. It tries to combine split expressions into deeply nested ones again and also improves the "compilability" for stack machines by eliminating variables as much as possible.

ExpressionJoiner

This is the opposite operation of the expression splitter. It turns a sequence of variable declarations that have exactly one reference into a complex expression. This stage fully preserves the order of function calls and opcode executions. It does not make use of any information concerning the commutativity of the opcodes; if moving the value of a variable to its place of use would change the order of any function call or opcode execution, the transformation is not performed.

Note that the component will not move the assigned value of a variable assignment or a variable that is referenced more than once.

The snippet `let x := add(0, 2) let y := mul(x, mload(2))` is not transformed, because it would cause the order of the call to the opcodes `add` and `mload` to be swapped - even though this would not make a difference because `add` is movable.

When reordering opcodes like that, variable references and literals are ignored. Because of that, the snippet `let x := add(0, 2) let y := mul(x, 3)` is transformed to `let y := mul(add(0, 2), 3)`, even though the `add` opcode would be executed after the evaluation of the literal 3.

SSAReverser

This is a tiny step that helps in reversing the effects of the SSA transform if it is combined with the Common Subexpression Eliminator and the Unused Pruner.

The SSA form we generate is detrimental to code generation on the EVM and WebAssembly alike because it generates many local variables. It would be better to just re-use existing variables with assignments instead of fresh variable declarations.

The SSA transform rewrites

```
let a := calldataload(0)
mstore(a, 1)
```

to

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

The problem is that instead of `a`, the variable `a_1` is used whenever `a` was referenced. The SSA transform changes statements of this form by just swapping out the declaration and the assignment. The above snippet is turned into

```
let a := calldataload(0)
let a_1 := a
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

This is a very simple equivalence transform, but when we now run the Common Subexpression Eliminator, it will replace all occurrences of `a_1` by `a` (until `a` is re-assigned). The Unused Pruner will then eliminate the variable `a_1` altogether and thus fully reverse the SSA transform.

StackCompressor

One problem that makes code generation for the Ethereum Virtual Machine hard is the fact that there is a hard limit of 16 slots for reaching down the expression stack. This more or less translates to a limit of 16 local variables. The stack compressor takes Yul code and compiles it to EVM bytecode. Whenever the stack difference is too large, it records the function this happened in.

For each function that caused such a problem, the Rematerialiser is called with a special request to aggressively eliminate specific variables sorted by the cost of their values.

On failure, this procedure is repeated multiple times.

Rematerialiser

The rematerialisation stage tries to replace variable references by the expression that was last assigned to the variable. This is of course only beneficial if this expression is comparatively cheap to evaluate. Furthermore, it is only semantically equivalent if the value of the expression did not change between the point of assignment and the point of use. The main benefit of this stage is that it can save stack slots if it leads to a variable being eliminated completely (see below), but it can also save a DUP opcode on the EVM if the expression is very cheap.

The Rematerialiser uses the Dataflow Analyzer to track the current values of variables, which are always movable. If the value is very cheap or the variable was explicitly requested to be eliminated, the variable reference is replaced by its current value.

ForLoopConditionOutOfBody

Reverses the transformation of ForLoopConditionIntoBody.

For any movable c, it turns

```
for { ... } l { ... } {
if iszero(c) { break }
...
}
```

into

```
for { ... } c { ... } {
...
}
```

and it turns

```
for { ... } l { ... } {
if c { break }
...
}
```

into

```
for { ... } iszero(c) { ... } {
...
}
```

The LiteralRematerialiser should be run before this step.

WebAssembly specific

MainFunction

Changes the topmost block to be a function with a specific name (“main”) which has no inputs nor outputs.

Depends on the Function Grouper.

3.21 Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the compiled contract. You can use this file to query the compiler version, the sources used, the ABI and NatSpec documentation to more safely interact with the contract and verify its source code.

The compiler appends by default the IPFS hash of the metadata file to the end of the bytecode (for details, see below) of each contract, so that you can retrieve the file in an authenticated way without having to resort to a centralized data provider. The other available options are the Swarm hash and not appending the metadata hash to the bytecode. These can be configured via the *Standard JSON Interface*.

You have to publish the metadata file to IPFS, Swarm, or another service so that others can access it. You create the file by using the `solc --metadata` command that generates a file called `ContractName_meta.json`. It contains IPFS and Swarm references to the source code, so you have to upload all source files and the metadata file.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum and sort the keys of all objects to arrive at a unique formatting. Comments are not permitted and used here only for explanatory purposes.

```
{
  // Required: The version of the metadata format
  "version": "1",
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  "language": "Solidity",
  // Required: Details about the compiler, contents are specific
  // to the language.
  "compiler": {
    // Required for Solidity: Version of the compiler
    "version": "0.4.6+commit.2dabbd0.Emscripten.clang",
    // Optional: Hash of the compiler binary which produced this output
    "keccak256": "0x123..."
  },
  // Required: Compilation source files/source units, keys are file names
  "sources":
  {
    "myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but a
      // Swarm URL is recommended
      "urls": [ "bzzr://56ab..." ],
      // Optional: SPDX license identifier as given in the source file
      "license": "MIT"
    },
    "destructible": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract destructible is owned { function destroy() { if (msg.sender == owner) selfdestruct(owner); } }"
    }
  },
  // Required: Compiler settings
  "settings":
  {
    // Required for Solidity: Sorted list of remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Optimizer settings. The fields "enabled" and "runs" are deprecated
    // and are only given for backwards-compatibility.
    "optimizer": {
      "enabled": true,
      "runs": 500,
      "details": {
        // peephole defaults to "true"
        "peephole": true,
        // inliner defaults to "true"
        "inliner": true
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    "inliner": true,
    // jumpdestRemover defaults to "true"
    "jumpdestRemover": true,
    "orderLiterals": false,
    "deduplicate": false,
    "cse": false,
    "constantOptimizer": false,
    "yul": true,
    // Optional: Only present if "yul" is "true"
    "yulDetails": {
        "stackAllocation": false,
        "optimizerSteps": "dhfoDgvulfnTUtnIf..."
    }
},
"metadata": {
    // Reflects the setting used in the input json, defaults to false
    "useLiteralContent": true,
    // Reflects the setting used in the input json, defaults to "ipfs"
    "bytecodeHash": "ipfs"
},
// Required for Solidity: File and name of the contract or library this
// metadata is created for.
"compilationTarget": {
    "myFile.sol": "MyContract"
},
// Required for Solidity: Addresses for libraries used
"libraries": {
    "MyLib": "0x123123..."
}
},
// Required: Generated information about the contract.
"output": {
    // Required: ABI definition of the contract
    "abi": [/* ... */],
    // Required: NatSpec user documentation of the contract
    "userdoc": [/* ... */],
    // Required: NatSpec developer documentation of the contract
    "devdoc": [/* ... */]
}
}
}

```

Warning: Since the bytecode of the resulting contract contains the metadata hash by default, any change to the metadata might result in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

Note: The ABI definition above has no fixed order. It can change with compiler versions. Starting from Solidity

version 0.5.12, though, the array maintains a certain order.

3.21.1 Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping {"ipfs": <IPFS hash>, "solc": <compiler version>} is stored CBOR-encoded. Since the mapping might contain more keys (see below) and the beginning of that encoding is not easy to find, its length is added in a two-byte big-endian encoding. The current version of the Solidity compiler usually adds the following to the end of the deployed bytecode

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 bytes IPFS hash>
0x64 's' 'o' 'l' 'c' 0x43 <3 byte version encoding>
0x00 0x33
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and use the IPFS hash to retrieve the file.

Whereas release builds of solc use a 3 byte encoding of the version as shown above (one byte each for major, minor and patch version number), prerelease builds will instead use a complete version string including commit hash and build date.

Note: The CBOR mapping can also contain other keys, so it is better to fully decode the data instead of relying on it starting with 0xa264. For example, if any experimental features that affect code generation are used, the mapping will also contain "experimental": true.

Note: The compiler currently uses the IPFS hash of the metadata by default, but it may also use the bzzr1 hash or some other hash in the future, so do not rely on this sequence to start with 0xa2 0x64 'i' 'p' 'f' 's'. We might also add additional data to this CBOR structure, so the best option is to use a proper CBOR parser.

3.21.2 Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way: A component that wants to interact with a contract (e.g. Mist or any wallet) retrieves the code of the contract, from that the IPFS/Swarm hash of a file which is then retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, the wallet can use the NatSpec user documentation to display a confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

For additional information, read [Ethereum Natural Language Specification \(NatSpec\) format](#).

3.21.3 Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from IPFS/Swarm via the link in the metadata file. The compiler of the correct version (which is checked to be part of the “official” compilers) is invoked on that input with the specified settings. The resulting bytecode is compared to the data of the creation transaction or CREATE opcode data. This automatically verifies the metadata since its hash is part of the bytecode. Excess data corresponds to the constructor input data, which should be decoded according to the interface and presented to the user.

In the repository [sourcify](#) ([npm package](#)) you can see example code that shows how to use this feature.

3.22 Contract ABI Specification

3.22.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time.

3.22.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Note: The return type of a function is not part of this signature. In *Solidity’s function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

3.22.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

3.22.4 Types

The following elementary types exist:

- **uint $<M>$:** unsigned integer type of M bits, $0 < M \leq 256$, $M \% 8 == 0$. e.g. uint32, uint8, uint256.
- **int $<M>$:** two’s complement signed integer type of M bits, $0 < M \leq 256$, $M \% 8 == 0$.
- **address:** equivalent to uint160, except for the assumed interpretation and language typing. For computing the function selector, address is used.

- `uint`, `int`: synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool`: equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>`: signed fixed-point decimal number of `M` bits, $8 \leq M \leq 256$, $M \% 8 == 0$, and $0 < N \leq 80$, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>`: unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed`: synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>`: binary type of `M` bytes, $0 < M \leq 32$.
- `function`: an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists:

- `<type>[M]`: a fixed-length array of `M` elements, $M \geq 0$, of the given type.

Note: While this ABI specification can express fixed-length arrays with zero elements, they're not supported by the compiler.

The following non-fixed-size types exist:

- `bytes`: dynamic sized byte sequence.
- `string`: dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]`: a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas:

- `(T1, T2, ..., Tn)`: tuple consisting of the types `T1, ..., Tn`, $n \geq 0$

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are not part of the ABI, and on the right column the ABI types that represent them.

Solidity	ABI
<code>address payable</code>	<code>address</code>
<code>contract</code>	<code>address</code>
<code>enum</code>	<code>uint8</code>
<code>user defined value types</code>	its underlying value type
<code>struct</code>	<code>tuple</code>

Warning: Before version 0.8.0 enums could have more than 256 members and were represented by the smallest integer type just big enough to hold the value of any member.

3.22.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][1][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative “addresses”.

3.22.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called “dynamic”:

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k >= 0`
- `(T1, ..., Tk)` if T_i is dynamic for some $1 \leq i \leq k$

All other types are called “static”.

Definition: `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition: For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1, ..., Tk)` for $k \geq 0$ and any types `T1, ..., Tk`

`enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for T_i as follows:

if T_i is static:

`head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

otherwise, i.e. if T_i is dynamic:

`head(X(i)) = enc(len(head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1))) tail(X(i)) = enc(X(i))`

Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. The value of `head(X(i))` is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:

`enc(X) = enc((X[0], ..., X[k-1]))`

i.e. it is encoded as if it were a tuple with `k` elements of the same type.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

```
enc(X) = enc(k) enc([X[0], ..., X[k-1]])
```

i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

`enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

`enc(X) = enc(enc_utf8(X))`, i.e. `X` is UTF-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the UTF-8 encoded string, not its number of characters.

- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.

- `address`: as in the `uint160` case

- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` bytes for negative `X` and with zero-bytes for non-negative `X` such that the length is 32 bytes.

- `bool`: as in the `uint8` case, where 1 is used for `true` and 0 for `false`

- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.

- `fixed`: as in the `fixed128x18` case

- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.

- `ufixed`: as in the `ufixed128x18` case

- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

3.22.7 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

3.22.8 Examples

Given the contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
```

(continues on next page)

(continued from previous page)

```
function sam(bytes memory, bool, uint[] memory) public pure {}  
}
```

Thus for our Foo example if we wanted to call baz with the parameters 69 and true, we would pass 68 bytes total, which can be broken down into:

In total:

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into:

In total:

If we wanted to call `sam` with the arguments "dave", `true` and `[1, 2, 3]`, we would pass 292 bytes total, broken down into:

In total:

3.22.9 Use of Dynamic Types

A call to a function with the signature `f(uint256,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

After this, the data part of the first dynamic argument, [0x456, 0x789] follows:

Finally, we encode the data part of the second dynamic argument, "Hello, world!":

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

(continues on next page)

(continued from previous page)

Let us apply the same principle to encode the data for a function with a signature `g(uint256[][], string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding:

First we encode the length and data of the first embedded dynamic array [1, 2] of the first root array [[1, 2], [3]]:

Then we encode the length and data of the second embedded dynamic array [3] of the first root array [[1, 2], [3]]:

Then we need to find the offsets a and b for their respective dynamic arrays [1, 2] and [3]. To calculate the offsets we can take a look at the encoded data of the first root array [[1, 2], [3]] enumerating each line in the encoding:

Then we encode the embedded strings of the second root array:

In parallel to the first root array, since strings are dynamic elements we need to find their offsets c , d and e :

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[], uint256[] [])`.

Then we encode the length of the first root array:

Then we encode the length of the second root array:

- `0x0003` (number of strings in the second root array, 3; the strings themselves are "one", "two" and "three")

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order:

(continues on next page)

(continued from previous page)

3.22.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3 (for non-anonymous events) or 4 (for anonymous ones), are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- address: the address of the contract (intrinsically provided by Ethereum);
 - topics[0]: keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")") (canonical_type_of is a function that simply returns the canonical type of a given argument, e.g. for uint indexed foo, it would return uint256). This value is only present in topics[0] if the event is not declared as anonymous;
 - topics[n]: abi_encode(EVENT_INDEXED_ARGS[n - 1]) if the event is not declared as anonymous or abi_encode(EVENT_INDEXED_ARGS[n]) if it is (EVENT_INDEXED_ARGS is the series of EVENT_ARGS that are indexed);
 - data: ABI encoding of EVENT_NON_INDEXED_ARGS (EVENT_NON_INDEXED_ARGS is the series of EVENT_ARGS that are not indexed, abi_encode is the ABI encoding function used for returning a series of typed values from a function, as described above).

For all types of length at most 32 bytes, the `EVENT_INDEXED_ARGS` array contains the value directly, padded or sign-extended (for signed integers) to 32 bytes, just as for regular ABI encoding. However, for all “complex” types or types of dynamic length, including all arrays, `string`, `bytes` and structs, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of a special in-place encoded value (see [Encoding of Indexed Event Parameters](#)), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

3.22.11 Errors

In case of a failure inside a contract, the contract can use a special opcode to abort execution and revert all state changes. In addition to these effects, descriptive data can be returned to the caller. This descriptive data is the encoding of an error and its arguments in the same way as data for a function call.

As an example, let us consider the following contract whose `transfer` function always reverts with a custom error of “insufficient balance”:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

The return data would be encoded in the same way as the function call `InsufficientBalance(0, amount)` to the function `InsufficientBalance(uint256, uint256)`, i.e. `0xcf479181, uint256(0), uint256(amount)`.

The error selectors `0x00000000` and `0xffffffff` are reserved for future use.

Warning: Never trust error data. The error data by default bubbles up through the chain of external calls, which means that a contract may receive an error not defined in any of the contracts it calls directly. Furthermore, any contract can fake any error by returning data that matches an error signature, even if the error is not defined anywhere.

3.22.12 JSON

The JSON format for a contract’s interface is given by an array of function, event and error descriptions. A function description is a JSON object with the fields:

- `type`: “`function`”, “`constructor`”, “`receive`” (the “*receive Ether* function”) or “`fallback`” (the “*default* function”);
- `name`: the name of the function;
- `inputs`: an array of objects, each of which contains:
 - `name`: the name of the parameter.
 - `type`: the canonical type of the parameter (more below).
 - `components`: used for tuple types (more below).
- `outputs`: an array of objects similar to `inputs`.
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether - the default) and `payable` (function accepts Ether).

Constructor and fallback function never have `name` or `outputs`. Fallback function doesn’t have `inputs` either.

Note: Sending non-zero Ether to non-payable function will revert the transaction.

Note: The state mutability `nonpayable` is reflected in Solidity by not specifying a state mutability modifier at all.

An event description is a JSON object with fairly similar fields:

- `type`: always "event"
- `name`: the name of the event.
- `inputs`: an array of objects, each of which contains:
 - `name`: the name of the parameter.
 - `type`: the canonical type of the parameter (more below).
 - `components`: used for tuple types (more below).
 - `indexed`: `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous`: `true` if the event was declared as `anonymous`.

Errors look as follows:

- `type`: always "error"
- `name`: the name of the error.
- `inputs`: an array of objects, each of which contains:
 - `name`: the name of the parameter.
 - `type`: the canonical type of the parameter (more below).
 - `components`: used for tuple types (more below).

Note: There can be multiple errors with the same name and even with identical signature in the JSON array, for example if the errors originate from different files in the smart contract or are referenced from another smart contract. For the ABI, only the name of the error itself is relevant and not where it is defined.

For example,

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

would result in the JSON:

```
[{
  "type": "error",
  "inputs": [{"name": "available", "type": "uint256"}, {"name": "required", "type": "uint256"}],
```

(continues on next page)

(continued from previous page)

```

"name": "InsufficientBalance"
}, {
"type": "event",
"inputs": [{"name": "a", "type": "uint256", "indexed": true}, {"name": "b", "type": "bytes32", "indexed": false}],
"name": "Event"
}, {
"type": "event",
"inputs": [{"name": "a", "type": "uint256", "indexed": true}, {"name": "b", "type": "bytes32", "indexed": false}],
"name": "Event2"
}, {
"type": "function",
"inputs": [{"name": "a", "type": "uint256"}],
"name": "foo",
"outputs": []
}
]

```

Handling tuple types

Despite that names are intentionally not part of the ABI encoding they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way:

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers k. The components of the tuple are then stored in the member `components`, which is of array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

would result in the JSON:

```
[
{
  "name": "f",
  "type": "function",
  "inputs": [
    {
      "name": "s",
      "type": "tuple",
      "components": [
        {
          "name": "a",
          "type": "uint"
        },
        {
          "name": "b",
          "type": "array",
          "components": [
            {
              "name": "x",
              "type": "uint"
            },
            {
              "name": "y",
              "type": "uint"
            }
          ]
        }
      ]
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
"components": [
    {
        "name": "a",
        "type": "uint256"
    },
    {
        "name": "b",
        "type": "uint256[]"
    },
    {
        "name": "c",
        "type": "tuple[]",
        "components": [
            {
                "name": "x",
                "type": "uint256"
            },
            {
                "name": "y",
                "type": "uint256"
            }
        ]
    },
    {
        "name": "t",
        "type": "tuple",
        "components": [
            {
                "name": "x",
                "type": "uint256"
            },
            {
                "name": "y",
                "type": "uint256"
            }
        ]
    },
    {
        "name": "a",
        "type": "uint256"
    }
],
"outputs": []
}]
```

3.22.13 Strict Encoding Mode

Strict encoding mode is the mode that leads to exactly the same encoding as defined in the formal specification above. This means offsets have to be as small as possible while still not creating overlaps in the data areas and thus no gaps are allowed.

Usually, ABI decoders are written in a straightforward way just following offset pointers, but some decoders might enforce strict mode. The Solidity ABI decoder currently does not enforce strict mode, but the encoder always creates data in strict mode.

3.22.14 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where:

- types shorter than 32 bytes are concatenated directly, without padding or sign extension
- dynamic types are encoded in-place and without the length.
- array elements are padded, but still encoded in-place

Furthermore, structs as well as nested arrays are not supported.

As an example, the encoding of `int16(-1)`, `bytes1(0x42)`, `uint16(0x03)`, `string("Hello, world!")` results in:

```
0xfffff42000348656c6c6f2c20776f726c6421
      ^^^^          int16(-1)
      ^^          bytes1(0x42)
      ^^^^          uint16(0x03)
      ^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically:

- During the encoding, everything is encoded in-place. This means that there is no distinction between head and tail, as in the ABI encoding, and the length of an array is not encoded.
- The direct arguments of `abi.encodePacked` are encoded without padding, as long as they are not arrays (or `string` or `bytes`).
- The encoding of an array is the concatenation of the encoding of its elements **with** padding.
- Dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field.
- The encoding of `string` or `bytes` does not apply padding at the end unless it is part of an array or struct (then it is padded to a multiple of 32 bytes).

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used: `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

Warning: If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. If you use `abi.encodePacked` for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, `abi.encode` should be preferred.

3.22.15 Encoding of Indexed Event Parameters

Indexed event parameters that are not value types, i.e. arrays and structs are not stored directly but instead a keccak256-hash of an encoding is stored. This encoding is defined as follows:

- the encoding of a `bytes` and `string` value is just the string contents without any padding or length prefix.
- the encoding of a struct is the concatenation of the encoding of its members, always padded to a multiple of 32 bytes (even `bytes` and `string`).
- the encoding of an array (both dynamically- and statically-sized) is the concatenation of the encoding of its elements, always padded to a multiple of 32 bytes (even `bytes` and `string`) and without any length prefix

In the above, as usual, a negative number is padded by sign extension and not zero padded. `bytesNN` types are padded on the right while `uintNN` / `intNN` are padded on the left.

Warning: The encoding of a struct is ambiguous if it contains more than one dynamically-sized array. Because of that, always re-check the event data and do not rely on the search result based on the indexed parameters alone.

3.23 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check [the release changelog](#).

Note: Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices. See the [Interoperability With Older Contracts](#) section below.

3.23.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.
- The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
- The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
- Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
- The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
- The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
- Forward all available gas with external function calls starting from Tangerine Whistle.

3.23.2 Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

- The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)")), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
- Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.
- Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

3.23.3 Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

- Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
- Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = z` to `uint[] storage x = z`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
- Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example: if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
- Explicit conversions between unrelated contract types are now disallowed. You can only convert from a contract type to one of its base or ancestor types. If you are sure that a contract is compatible with the contract type you want to convert to, although it does not inherit from it, you can work around this by converting to `address` first. Example: if `A` and `B` are contract types, `B` does not inherit from `A` and `b` is a contract of type `B`, you can still convert `b` to type `A` using `A(address(b))`. Note that you still need to watch out for matching payable fallback functions, as explained below.
- The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the [withdraw pattern](#), you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
- Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes)

by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`. Before v0.5.0 any conversion between `bytesX` and `uintY` would go through `uint8X`. For example `uint8(bytes3(0x291807))` would be converted to `uint8(uint24(bytes3(0x291807)))` (the result is `0x07`).

- Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
- For clarity reasons, the command line interface now requires – if the standard input is used as source.

3.23.4 Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

Command Line and JSON Interfaces

- The command line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
- The command line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language Julia to Yul.
- The `--clone-bin` and `--combined-json clone-bin` command line options were removed.
- Remappings with empty prefix are disallowed.
- The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
- The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value "constructor", "fallback" or "function".
- In unlinked binary hex files, library address placeholders are now the first 36 hex characters of the keccak256 hash of the fully qualified library name, surrounded by `$...$`. Previously, just the fully qualified library name was used. This reduces the chances of collisions, especially when long paths are used. Binary files now also contain a list of mappings from these placeholders to the fully qualified names.

Constructors

- Constructors must now be defined using the `constructor` keyword.
- Calling base constructors without parentheses is now disallowed.
- Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
- Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

Functions

- Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
- `suicide` is now disallowed (in favor of `selfdestruct`).
- `sha3` is now disallowed (in favor of `keccak256`).
- `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

Conversions

- Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
- Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

Literals and Suffixes

- The unit denomination `years` is now disallowed due to complications and confusions about leap years.
- Trailing dots that are not followed by a number are now disallowed.
- Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
- The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

Variables

- Declaring empty structs is now disallowed for clarity.
- The `var` keyword is now disallowed to favor explicitness.
- Assignments between tuples with different number of components is now disallowed.
- Values for constants that are not compile-time constants are disallowed.
- Multi-variable declarations with mismatching number of values are now disallowed.
- Uninitialized storage variables are now disallowed.
- Empty tuple components are now disallowed.
- Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
- Fixed-size arrays with a length of zero are now disallowed.

Syntax

- Using `constant` as function state mutability modifier is now disallowed.
- Boolean expressions cannot use arithmetic operations.
- The unary `+` operator is now disallowed.
- Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
- Empty return statements for functions with one or more return values are now disallowed.
- The “loose assembly” syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.

- Functions without implementation cannot use modifiers anymore.
- Function types with named return values are now disallowed.
- Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
- New keywords: `calldata` and `constructor`.
- New reserved keywords: `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

3.23.5 Interoperability With Older Contracts

It is still possible to interface with contracts written for Solidity versions prior to v0.5.0 (or the other way around) by defining interfaces for them. Consider you have the following pre-0.5.0 contract already deployed:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will report a warning until version 0.4.25 of the compiler
// This will not compile after 0.5.0
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

This will no longer compile with Solidity v0.5.0. However, you can define a compatible interface for it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Note that we did not declare `anotherOldFunction` to be `view`, despite it being declared `constant` in the original contract. This is due to the fact that starting with Solidity v0.5.0 `staticcall` is used to call `view` functions. Prior to v0.5.0 the `constant` keyword was not enforced, so calling a function declared `constant` with `staticcall` may still revert, since the `constant` function may still attempt to modify storage. Consequently, when defining an interface for older contracts, you should only use `view` in place of `constant` in case you are absolutely sure that the function will work with `staticcall`.

Given the interface defined above, you can now easily use the already deployed pre-0.5.0 contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

(continues on next page)

(continued from previous page)

```
contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Similarly, pre-0.5.0 libraries can be used by defining the functions of the library without implementation and supplying the address of the pre-0.5.0 library during linking (see [Using the Commandline Compiler](#) for how to use the commandline compiler for linking):

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.23.6 Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will not compile after 0.5.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }
}
```

(continues on next page)

(continued from previous page)

```

// Function visibility not provided, not an error.
// Function mutability not provided, not an error.
function f(uint x) returns (bytes) {
    // Var is fine in this version.
    var z = someInteger();
    x += z;
    // Throw is fine in this version.
    if (x > 100)
        throw;
    bytes memory b = new bytes(x);
    y = -3 >> 1;
    // y == -1 (wrong, should be -2)
    do {
        x += 1;
        if (x > 10) continue;
        // 'Continue' causes an infinite loop.
    } while (x < 11);
    // Call returns only a Bool.
    bool success = address(other).call("f");
    if (!success)
        revert();
    else {
        // Local variables could be declared after their use.
        int y;
    }
    return b;
}

// No need for an explicit data location for 'arr'
function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
    otherContract.transfer(1 ether);

    // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
    // the first 4 bytes of x will be lost. This might lead to
    // unexpected behavior since bytesX are right padded.
    uint32 y = uint32(x);
    myNumber += y + msg.value;
}
}

```

New version:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// This will not compile after 0.6.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
}

```

(continues on next page)

(continued from previous page)

```

function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using AddressMakePayable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract, address
    ↴unknownContract) public payable {
        // 'otherContract.transfer' is not provided.
        // Since the code of 'OtherContract' is known and has the fallback
        // function, address(otherContract) has type 'address payable'.
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' is not provided.
        // 'address(unknownContract).transfer' is not provided
        // since 'address(unknownContract)' is not 'address payable'.
        // If the function takes an 'address' which you want to send
        // funds to, you can convert it to 'address payable' via 'uint160'.
        // Note: This is not recommended and the explicit type
        // 'address payable' should be used whenever possible.
        // To increase clarity, we suggest the use of a library for
        // the conversion (provided after the contract in this example).
    }
}

```

(continues on next page)

(continued from previous page)

```

address payable addr = unknownContract.makePayable();
require(addr.send(1 ether));

// Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
// the conversion is not allowed.
// We need to convert to a common size first:
bytes4 x4 = bytes4(x); // Padding happens on the right
uint32 y = uint32(x4); // Conversion is consistent
// 'msg.value' cannot be used in a 'non-payable' function.
// We need to make the function payable
myNumber += y + msg.value;
}

}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library AddressMakePayable {
    function makePayable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}

```

3.24 Solidity v0.6.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.6.0, along with the reasoning behind the changes and how to update affected code. For the full list check [the release changelog](#).

3.24.1 Changes the Compiler Might not Warn About

This section lists changes where the behaviour of your code might change without the compiler telling you about it.

- The resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmetric operations. Additionally, signed types are allowed for the base of the exponentiation.

3.24.2 Explicitness Requirements

This section lists changes where the code now needs to be more explicit, but the semantics do not change. For most of the topics the compiler will provide suggestions.

- Functions can now only be overridden when they are either marked with the `virtual` keyword or defined in an interface. Functions without implementation outside an interface have to be marked `virtual`. When overriding a function or modifier, the new keyword `override` must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so: `override(Base1, Base2)`.
- Member-access to `length` of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their `length`. Use `push()`, `push(value)` or `pop()` instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.

- The new keyword `abstract` can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions. Abstract contracts cannot be created using the `new` operator, and it is not possible to generate bytecode for them during compilation.
- Libraries have to implement all their functions, not only the internal ones.
- The names of variables declared in inline assembly may no longer end in `_slot` or `_offset`.
- Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.
- State variable shadowing is now disallowed. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

3.24.3 Semantic and Syntactic Changes

This section lists changes where you have to modify your code and it does something else afterwards.

- Conversions from external function types to `address` are now disallowed. Instead external function types have a member called `address`, similar to the existing `selector` member.
- The function `push(value)` for dynamic storage arrays does not return the new length anymore (it returns nothing).
- The unnamed function commonly referred to as “fallback function” was split up into a new fallback function that is defined using the `fallback` keyword and a receive ether function defined using the `receive` keyword.
 - If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). This function is implicitly `payable`.
 - The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty call data). You can make this function `payable` or not. If it is not `payable` then transactions not matching any other function which send value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

3.24.4 New Features

This section lists things that were not possible prior to Solidity 0.6.0 or were more difficult to achieve.

- The `try/catch statement` allows you to react on failed external calls.
- `struct` and `enum` types can be declared at file level.
- Array slices can be used for calldata arrays, for example `abi.decode(msg.data[4:], (uint, uint))` is a low-level way to decode the function call payload.
- Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as `@param`.
- Yul and Inline Assembly have a new statement called `leave` that exits the current function.
- Conversions from `address` to `address payable` are now possible via `payable(x)`, where `x` must be of type `address`.

3.24.5 Interface Changes

This section lists changes that are unrelated to the language itself, but that have an effect on the interfaces of the compiler. These may change the way how you use the compiler on the command line, how you use its programmable interface, or how you analyze the output produced by it.

New Error Reporter

A new error reporter was introduced, which aims at producing more accessible error messages on the command line. It is enabled by default, but passing `--old-reporter` falls back to the the deprecated old error reporter.

Metadata Hash Options

The compiler now appends the [IPFS](#) hash of the metadata file to the end of the bytecode by default (for details, see documentation on [contract metadata](#)). Before 0.6.0, the compiler appended the [Swarm](#) hash by default, and in order to still support this behaviour, the new command line option `--metadata-hash` was introduced. It allows you to select the hash to be produced and appended, by passing either `ipfs` or `swarm` as value to the `--metadata-hash` command line option. Passing the value `none` completely removes the hash.

These changes can also be used via the [Standard JSON Interface](#) and effect the metadata JSON generated by the compiler.

The recommended way to read the metadata is to read the last two bytes to determine the length of the CBOR encoding and perform a proper decoding on that data block as explained in the [metadata section](#).

Yul Optimizer

Together with the legacy bytecode optimizer, the [Yul](#) optimizer is now enabled by default when you call the compiler with `--optimize`. It can be disabled by calling the compiler with `--no-optimize-yul`. This mostly affects code that uses ABI coder v2.

C API Changes

The client code that uses the C API of `libsolc` is now in control of the memory used by the compiler. To make this change consistent, `solidity_free` was renamed to `solidity_reset`, the functions `solidity_alloc` and `solidity_free` were added and `solidity_compile` now returns a string that must be explicitly freed via `solidity_free()`.

3.24.6 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `address(f)` to `f.address` for `f` being of external function type.
- Replace `function () external [payable] { ... }` by either `receive() external payable { ... }`, `fallback() external [payable] { ... }` or both. Prefer using a `receive` function only, whenever possible.
- Change `uint length = array.push(value)` to `array.push(value);`. The new length can be accessed via `array.length`.
- Change `array.length++` to `array.push()` to increase, and use `pop()` to decrease the length of a storage array.

- For every named return parameter in a function's `@dev` documentation define a `@return` entry which contains the parameter's name as the first word. E.g. if you have function `f()` defined like `function f() public returns (uint value)` and a `@dev` annotating it, document its return parameters like so: `@return value The return value..` You can mix named and un-named return parameters documentation so long as the notices are in the order they appear in the tuple return type.
- Choose unique identifiers for variable declarations in inline assembly that do not conflict with declarations outside the inline assembly block.
- Add `virtual` to every non-interface function you intend to override. Add `virtual` to all functions without implementation outside interfaces. For single inheritance, add `override` to every overriding function. For multiple inheritance, add `override(A, B, ...)`, where you list all contracts that define the overridden function in the parentheses. When multiple bases define the same function, the inheriting contract must override all conflicting functions.

3.25 Solidity v0.7.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.7.0, along with the reasoning behind the changes and how to update affected code. For the full list check the [release changelog](#).

3.25.1 Silent Changes of the Semantics

- Exponentiation and shifts of literals by non-literals (e.g. `1 << x` or `2 ** x`) will always use either the type `uint256` (for non-negative literals) or `int256` (for negative literals) to perform the operation. Previously, the operation was performed in the type of the shift amount / the exponent which can be misleading.

3.25.2 Changes to the Syntax

- In external function and contract creation calls, Ether and gas is now specified using a new syntax: `x.f{gas: 10000, value: 2 ether}(arg1, arg2)`. The old syntax – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – will cause an error.
- The global variable `now` is deprecated, `block.timestamp` should be used instead. The single identifier `now` is too generic for a global variable and could give the impression that it changes during transaction processing, whereas `block.timestamp` correctly reflects the fact that it is just a property of the block.
- NatSpec comments on variables are only allowed for public state variables and not for local or internal variables.
- The token `gwei` is a keyword now (used to specify, e.g. `2 gwei` as a number) and cannot be used as an identifier.
- String literals now can only contain printable ASCII characters and this also includes a variety of escape sequences, such as hexadecimal (`\xff`) and unicode escapes (`\u20ac`).
- Unicode string literals are supported now to accommodate valid UTF-8 sequences. They are identified with the `unicode` prefix: `unicode"Hello "`.
- State Mutability: The state mutability of functions can now be restricted during inheritance: Functions with default state mutability can be overridden by `pure` and `view` functions while `view` functions can be overridden by `pure` functions. At the same time, public state variables are considered `view` and even `pure` if they are constants.

Inline Assembly

- Disallow `.` in user-defined function and variable names in inline assembly. It is still valid if you use Solidity in Yul-only mode.
- Slot and offset of storage pointer variable `x` are accessed via `x.slot` and `x.offset` instead of `x_slot` and `x_offset`.

3.25.3 Removal of Unused or Unsafe Features

Mappings outside Storage

- If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.
- Assignments to structs or arrays in storage does not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.

Functions and Events

- Visibility (`public` / `internal`) is not needed for constructors anymore: To prevent a contract from being created, it can be marked `abstract`. This makes the visibility concept for constructors obsolete.
- Type Checker: Disallow `virtual` for library functions: Since libraries cannot be inherited from, library functions should not be virtual.
- Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.
- `using A for B` only affects the contract it is mentioned in. Previously, the effect was inherited. Now, you have to repeat the `using` statement in all derived contracts that make use of the feature.

Expressions

- Shifts by signed types are disallowed. Previously, shifts by negative amounts were allowed, but reverted at runtime.
- The `finney` and `szabo` denominations are removed. They are rarely used and do not make the actual amount readily visible. Instead, explicit values like `1e20` or the very common `gwei` can be used.

Declarations

- The keyword `var` cannot be used anymore. Previously, this keyword would parse but result in a type error and a suggestion about which type to use. Now, it results in a parser error.

3.25.4 Interface Changes

- JSON AST: Mark hex string literals with kind: "hexString".
- JSON AST: Members with value `null` are removed from JSON output.
- NatSpec: Constructors and functions have consistent userdoc output.

3.25.5 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `x.f.value(...)` to `x.f{value: ...}`. Similarly `(new C).value(...)` to `new C{value: ...}` and `x.f.gas(...).value(...)` to `x.f{gas: ..., value: ...}`.
- Change now to `block.timestamp`.
- Change types of right operand in shift operators to unsigned types. For example change `x >> (256 - y)` to `x >> uint(256 - y)`.
- Repeat the `using A for B` statements in all derived contracts if needed.
- Remove the `public` keyword from every constructor.
- Remove the `internal` keyword from every constructor and add `abstract` to the contract (if not already present).
- Change `_slot` and `_offset` suffixes in inline assembly to `.slot` and `.offset`, respectively.

3.26 Solidity v0.8.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.8.0. For the full list check [the release changelog](#).

3.26.1 Silent Changes of the Semantics

This section lists changes where existing code changes its behaviour without the compiler notifying you about it.

- Arithmetic operations revert on underflow and overflow. You can use `unchecked { ... }` to use the previous wrapping behaviour.

Checks for overflow are very common, so we made them the default to increase readability of code, even if it comes at a slight increase of gas costs.

- ABI coder v2 is activated by default.

You can choose to use the old behaviour using `pragma abicoder v1;`. The pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect. If you want to be explicit, please use `pragma abicoder v2;` instead.

Note that ABI coder v2 supports more types than v1 and performs more sanity checks on the inputs. ABI coder v2 makes some function calls more expensive and it can also make contract calls revert that did not revert with ABI coder v1 when they contain data that does not conform to the parameter types.

- Exponentiation is right associative, i.e., the expression `a**b**c` is parsed as `a**(b**c)`. Before 0.8.0, it was parsed as `(a**b)**c`.

This is the common way to parse the exponentiation operator.

- Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the invalid opcode but instead the revert opcode. More specifically, they will use error data equal to a function call to `Panic(uint256)` with an error code specific to the circumstances.

This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing `require`.

- If a byte array in storage is accessed whose length is encoded incorrectly, a panic is caused. A contract cannot get into this situation unless inline assembly is used to modify the raw representation of storage byte arrays.
- If constants are used in array length expressions, previous versions of Solidity would use arbitrary precision in all branches of the evaluation tree. Now, if constant variables are used as intermediate expressions, their values will be properly rounded in the same way as when they are used in run-time expressions.
- The type `byte` has been removed. It was an alias of `bytes1`.

3.26.2 New Restrictions

This section lists changes that might cause existing contracts to not compile anymore.

- There are new restrictions related to explicit conversions of literals. The previous behaviour in the following cases was likely ambiguous:
 1. Explicit conversions from negative literals and literals larger than `type(uint160).max` to `address` are disallowed.
 2. Explicit conversions between literals and an integer type `T` are only allowed if the literal lies between `type(T).min` and `type(T).max`. In particular, replace usages of `uint(-1)` with `type(uint).max`.
 3. Explicit conversions between literals and enums are only allowed if the literal can represent a value in the enum.
 4. Explicit conversions between literals and `address` type (e.g. `address(literal)`) have the type `address` instead of `address payable`. One can get a payable address type by using an explicit conversion, i.e., `payable(literal)`.
- *Address literals* have the type `address` instead of `address payable`. They can be converted to `address payable` by using an explicit conversion, e.g. `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.
- There are new restrictions on explicit type conversions. The conversion is only allowed when there is at most one change in sign, width or type-category (`int`, `address`, `bytesNN`, etc.). To perform multiple changes, use multiple conversions.

Let us use the notation `T(S)` to denote the explicit conversion `T(x)`, where, `T` and `S` are types, and `x` is any arbitrary variable of type `S`. An example of such a disallowed conversion would be `uint16(int8)` since it changes both width (8 bits to 16 bits) and sign (signed integer to unsigned integer). In order to do the conversion, one has to go through an intermediate type. In the previous example, this would be `uint16(uint8(int8))` or `uint16(int16(int8))`. Note that the two ways to convert will produce different results e.g., for `-1`. The following are some examples of conversions that are disallowed by this rule.

- `address(uint)` and `uint(address)`: converting both type-category and width. Replace this by `address(uint160(uint))` and `uint(uint160(address))` respectively.
- `payable(uint160)`, `payable(bytes20)` and `payable(integer-literal)`: converting both type-category and state-mutability. Replace this by `payable(address(uint160))`, `payable(address(bytes20))` and `payable(address(integer-literal))` respectively. Note that `payable(0)` is valid and is an exception to the rule.
- `int80(bytes10)` and `bytes10(int80)`: converting both type-category and sign. Replace this by `int80(uint80(bytes10))` and `bytes10(uint80(int80))` respectively.

- `Contract(uint)`: converting both type-category and width. Replace this by `Contract(address(uint160(uint)))`.

These conversions were disallowed to avoid ambiguity. For example, in the expression `uint16 x = uint16(int8(-1))`, the value of `x` would depend on whether the sign or the width conversion was applied first.

- Function call options can only be given once, i.e. `c.f{gas: 10000}{value: 1}()` is invalid and has to be changed to `c.f{gas: 10000, value: 1}()`.
- The global functions `log0`, `log1`, `log2`, `log3` and `log4` have been removed.

These are low-level functions that were largely unused. Their behaviour can be accessed from inline assembly.

- `enum` definitions cannot contain more than 256 members.

This will make it safe to assume that the underlying type in the ABI is always `uint8`.

- Declarations with the name `this`, `super` and `_` are disallowed, with the exception of public functions and events. The exception is to make it possible to declare interfaces of contracts implemented in languages other than Solidity that do permit such function names.
- Remove support for the `\b`, `\f`, and `\v` escape sequences in code. They can still be inserted via hexadecimal escapes, e.g. `\x08`, `\x0c`, and `\x0b`, respectively.
- The global variables `tx.origin` and `msg.sender` have the type `address` instead of `address payable`. One can convert them into `address payable` by using an explicit conversion, i.e., `payable(tx.origin)` or `payable(msg.sender)`.

This change was done since the compiler cannot determine whether or not these addresses are payable or not, so it now requires an explicit conversion to make this requirement visible.

- Explicit conversion into `address` type always returns a non-payable `address` type. In particular, the following explicit conversions have the type `address` instead of `address payable`:
 - `address(u)` where `u` is a variable of type `uint160`. One can convert `u` into the type `address payable` by using two explicit conversions, i.e., `payable(address(u))`.
 - `address(b)` where `b` is a variable of type `bytes20`. One can convert `b` into the type `address payable` by using two explicit conversions, i.e., `payable(address(b))`.
 - `address(c)` where `c` is a contract. Previously, the return type of this conversion depended on whether the contract can receive Ether (either by having a `receive` function or a `payable` fallback function). The conversion `payable(c)` has the type `address payable` and is only allowed when the contract `c` can receive Ether. In general, one can always convert `c` into the type `address payable` by using the following explicit conversion: `payable(address(c))`. Note that `address(this)` falls under the same category as `address(c)` and the same rules apply for it.
- The `chainid` builtin in inline assembly is now considered `view` instead of `pure`.
- Unary negation cannot be used on unsigned integers anymore, only on signed integers.

3.26.3 Interface Changes

- The output of `--combined-json` has changed: JSON fields `abi`, `devdoc`, `userdoc` and `storage-layout` are sub-objects now. Before 0.8.0 they used to be serialised as strings.
- The “legacy AST” has been removed (`--ast-json` on the commandline interface and `legacyAST` for standard JSON). Use the “compact AST” (`--ast-compact--json` resp. `AST`) as replacement.
- The old error reporter (`--old-reporter`) has been removed.

3.26.4 How to update your code

- If you rely on wrapping arithmetic, surround each operation with `unchecked { ... }`.
- Optional: If you use SafeMath or a similar library, change `x.add(y)` to `x + y`, `x.mul(y)` to `x * y` etc.
- Add `pragma abicoder v1`; if you want to stay with the old ABI coder.
- Optionally remove `pragma experimental ABIEncoderV2` or `pragma abicoder v2` since it is redundant.
- Change `byte` to `bytes1`.
- Add intermediate explicit type conversions if required.
- Combine `c.f{gas: 10000}{value: 1}()` to `c.f{gas: 10000, value: 1}()`.
- Change `msg.sender.transfer(x)` to `payable(msg.sender).transfer(x)` or use a stored variable of address `payable` type.
- Change `x**y**z` to `(x**y)**z`.
- Use inline assembly as a replacement for `log0, ..., log4`.
- Negate unsigned integers by subtracting them from the maximum value of the type and adding 1 (e.g. `type(uint256).max - x + 1`, while ensuring that `x` is not zero)

3.27 NatSpec Format

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

Note: NatSpec was inspired by [Doxygen](#). While it uses Doxygen-style comments and tags, there is no intention to keep strict compatibility with Doxygen. Please carefully examine the supported tags listed below.

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler. Also detailed below is output of the Solidity compiler, which extracts these comments into a machine-readable format.

NatSpec may also include annotations used by third-party tools. These are most likely accomplished via the `@custom:<name>` tag, and a good use case is analysis and verification tools.

3.27.1 Documentation Example

Documentation is inserted above each contract, interface, library, function, and event using the Doxygen notation format. A `public` state variable is equivalent to a function for the purposes of NatSpec.

- For Solidity you may choose `///` for single or multi-line comments, or `/**` and ending with `*/`.
- For Vyper, use `"""` indented to the inner contents with bare comments. See the [Vyper documentation](#).

The following example shows a contract and a function using all available tags.

Note: The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

This may change in the future.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
/// @custom:experimental This is an experimental contract.
contract Tree {
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Teteearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return Age in years, rounded up for partial years
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Returns the amount of leaves the tree has.
    /// @dev Returns only a fixed number.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}

contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// Return the amount of leaves that this specific kind of tree has
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure returns(uint256) {
```

(continues on next page)

(continued from previous page)

```

    return 3;
}
}

```

3.27.2 Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used. As a special case, if no tags are used then the Solidity compiler will interpret a `///` or `/**` comment in the same way as if it were tagged with `@notice`.

Tag		Context
<code>@title</code>	A title that should describe the contract/interface	contract, library, interface
<code>@author</code>	The name of the author	contract, library, interface
<code>@notice</code>	Explain to an end user what this does	contract, library, interface, function, public state variable, event
<code>@dev</code>	Explain to a developer any extra details	contract, library, interface, function, state variable, event
<code>@param</code>	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event
<code>@return</code>	Documents the return variables of a contract's function	function, public state variable
<code>@inherited</code>	Copies all missing tags from the base function (must be followed by the contract name)	function, public state variable
<code>@custom:</code> ..	Custom tag, semantics is application-defined	everywhere

If your function returns multiple values, like `(int quotient, int remainder)` then use multiple `@return` statements in the same format as the `@param` statements.

Custom tags start with `@custom:` and must be followed by one or more lowercase letters or hyphens. It cannot start with a hyphen however. They can be used everywhere and are part of the developer documentation.

Dynamic expressions

The Solidity compiler will pass through NatSpec documentation from your Solidity source code to the JSON output as described in this guide. The consumer of this JSON output, for example the end-user client software, may present this to the end-user directly or it may apply some pre-processing.

For example, some client software will render:

```
/// @notice This function will multiply `a` by 7
```

to the end-user as:

```
This function will multiply 10 by 7
```

if a function is being called and the input `a` is assigned a value of 10.

Specifying these dynamic expressions is outside the scope of the Solidity documentation and you may read more at [the radspec project](#).

Inheritance Notes

Functions without NatSpec will automatically inherit the documentation of their base function. Exceptions to this are:

- When the parameter names are different.
- When there is more than one base function.
- When there is an explicit `@inheritdoc` tag which specifies which contract should be used to inherit.

3.27.3 Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON files. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `ex1.sol` then you can generate the documentation using:

```
solc --userdoc --devdoc ex1.sol
```

And the output is below.

Note: Starting Solidity version 0.6.11 the NatSpec output also contains a `version` and a `kind` field. Currently the `version` is set to 1 and `kind` must be one of `user` or `dev`. In the future it is possible that new versions will be introduced, deprecating older ones.

User Documentation

The above documentation will produce the following user documentation JSON file as output:

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function's canonical signature as defined in the [Contract ABI](#) and not simply the function's name.

Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this:

```
{  
    "version" : 1,  
    "kind" : "dev",  
    "author" : "Larry A. Gardner",  
    "details" : "All function calls are currently implemented without side effects",  
    "custom.experimental" : "This is an experimental contract.",  
    "methods" :  
    {  
        "age(uint256)" :  
        {  
            "details" : "The Alexandr N. Teteearing algorithm could increase precision",  
            "params" :  
            {  
                "rings" : "The number of rings from dendrochronological sample"  
            },  
            "return" : "age in years, rounded up for partial years"  
        }  
    },  
    "title" : "A simulator for trees"  
}
```

3.28 Security Considerations

While it is usually quite easy to build software that works as expected, it is much harder to check that nobody can use it in a way that was **not** anticipated.

In Solidity, this is even more important because you can use smart contracts to handle tokens or, possibly, even more valuable things. Furthermore, every execution of a smart contract happens in public and, in addition to that, the source code is often available.

Of course you always have to consider how much is at stake: You can compare a smart contract with a web service that is open to the public (and thus, also to malicious actors) and perhaps even open source. If you only store your grocery list on that web service, you might not have to take too much care, but if you manage your bank account using that web service, you should be more careful.

This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug. A list of some publicly known security-relevant bugs of the compiler can be found in the [list of known bugs](#), which is also machine-readable. Note that there is a bug bounty program that covers the code generator of the Solidity compiler.

As always, with open source documentation, please help us extend this section (especially, some examples would not hurt)!

NOTE: In addition to the list below, you can find more security recommendations and best practices in Guy Lando's knowledge list and the Consensys GitHub repo.

3.28.1 Pitfalls

Private Information and Randomness

Everything you use in a smart contract is publicly visible, even local variables and state variables marked `private`.

Using random numbers in smart contracts is quite tricky if you do not want miners to be able to cheat.

Re-Entrancy

Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed. To give an example, the following code contains a bug (it is just a snippet and not a complete contract):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

The problem is not too serious here because of the limited gas as part of `send`, but it still exposes a weakness: Ether transfer can always include code execution, so the recipient could be a contract that calls back into `withdraw`. This would let it get multiple refunds and basically retrieve all the Ether in the contract. In particular, the following contract will allow an attacker to refund multiple times as it uses `call` which forwards all remaining gas by default:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

To avoid re-entrancy, you can use the Checks-Effects-Interactions pattern as outlined further below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;
```

(continues on next page)

(continued from previous page)

```
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

Note that re-entrancy is not only an effect of Ether transfer but of any function call on another contract. Furthermore, you also have to take multi-contract situations into account. A called contract could modify the state of another contract you depend on.

Gas Limit and Loops

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to `view` functions that are only executed to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

Sending and Receiving Ether

- Neither contracts nor “external accounts” are currently able to prevent that someone sends them Ether. Contracts can react on and reject a regular transfer, but there are ways to move Ether without creating a message call. One way is to simply “mine to” the contract address and the second way is using `selfdestruct(x)`.
- If a contract receives Ether (without a function being called), either the `receive Ether` or the `fallback` function is executed. If it does not have a receive nor a fallback function, the Ether will be rejected (by throwing an exception). During the execution of one of these functions, the contract can only rely on the “gas stipend” it is passed (2300 gas) being available to it at that time. This stipend is not enough to modify storage (do not take this for granted though, the stipend might change with future hard forks). To be sure that your contract can receive Ether in that way, check the gas requirements of the receive and fallback functions (for example in the “details” section in Remix).
- There is a way to forward more gas to the receiving contract using `addr.call{value: x}("")`. This is essentially the same as `addr.transfer(x)`, only that it forwards all remaining gas and opens up the ability for the recipient to perform more expensive actions (and it returns a failure code instead of automatically propagating the error). This might include calling back into the sending contract or other state changes you might not have thought of. So it allows for great flexibility for honest users but also for malicious actors.
- Use the most precise units to represent the wei amount as possible, as you lose any that is rounded due to a lack of precision.
- If you want to send Ether using `address.transfer`, there are certain details to be aware of:
 1. If the recipient is a contract, it causes its receive or fallback function to be executed which can, in turn, call back the sending contract.
 2. Sending Ether can fail due to the call depth going above 1024. Since the caller is in total control of the call depth, they can force the transfer to fail; take this possibility into account or use `send` and make sure

to always check its return value. Better yet, write your contract using a pattern where the recipient can withdraw Ether instead.

3. Sending Ether can also fail because the execution of the recipient contract requires more than the allotted amount of gas (explicitly by using `require`, `assert`, `revert` or because the operation is too expensive) - it “runs out of gas” (OOG). If you use `transfer` or `send` with a return value check, this might provide a means for the recipient to block progress in the sending contract. Again, the best practice here is to use a “*withdraw*” pattern instead of a “*send*” pattern.

Call Stack Depth

External function calls can fail any time because they exceed the maximum call stack size limit of 1024. In such situations, Solidity throws an exception. Malicious actors might be able to force the call stack to a high value before they interact with your contract. Note that, since Tangerine Whistle hardfork, the 63/64 rule makes call stack depth attack impractical. Also note that the call stack and the expression stack are unrelated, even though both have a size limit of 1024 stack slots.

Note that `.send()` does **not** throw an exception if the call stack is depleted but rather returns `false` in that case. The low-level functions `.call()`, `.delegatecall()` and `.staticcall()` behave in the same way.

Authorized Proxies

If your contract can act as a proxy, i.e. if it can call arbitrary contracts with user-supplied data, then the user can essentially assume the identity of the proxy contract. Even if you have other protective measures in place, it is best to build your contract system such that the proxy does not have any permissions (not even for itself). If needed, you can accomplish that using a second proxy:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function call0Other(address addr, bytes memory payload) public
        returns (bool, bytes memory)
    {
        return proxy.call0Other(addr, payload);
    }
    // Other functions and other functionality
}

// This is the full contract, it has no other functionality and
// requires no privileges to work.
contract PermissionlessProxy {
    function call0Other(address addr, bytes memory payload) public
        returns (bool, bytes memory)
    {
        return addr.call(payload);
    }
}
```

tx.origin

Never use tx.origin for authorization. Let's say you have a wallet contract like this:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        // THE BUG IS RIGHT HERE, you must use msg.sender instead of tx.origin
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Now someone tricks you into sending Ether to the address of this attack wallet:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

If your wallet had checked `msg.sender` for authorization, it would get the address of the attack wallet, instead of the owner address. But by checking `tx.origin`, it gets the original address that kicked off the transaction, which is still the owner address. The attack wallet instantly drains all your funds.

Two's Complement / Underflows / Overflows

As in many programming languages, Solidity's integer types are not actually integers. They resemble integers when the values are small, but cannot represent arbitrarily large numbers.

The following code causes an overflow because the result of the addition is too large to be stored in the type `uint8`:

```
uint8 x = 255;
uint8 y = 1;
return x + y;
```

Solidity has two modes in which it deals with these overflows: Checked and Unchecked or “wrapping” mode.

The default checked mode will detect overflows and cause a failing assertion. You can disable this check using `unchecked { ... }`, causing the overflow to be silently ignored. The above code would return `0` if wrapped in `unchecked { ... }`.

Even in checked mode, do not assume you are protected from overflow bugs. In this mode, overflows will always revert. If it is not possible to avoid the overflow, this can lead to a smart contract being stuck in a certain state.

In general, read about the limits of two's complement representation, which even has some more special edge cases for signed numbers.

Try to use `require` to limit the size of inputs to a reasonable range and use the *SMT checker* to find potential overflows.

Clearing Mappings

The Solidity type `mapping` (see [Mapping Types](#)) is a storage-only key-value data structure that does not keep track of the keys that were assigned a non-zero value. Because of that, cleaning a mapping without extra information about the written keys is not possible. If a `mapping` is used as the base type of a dynamic storage array, deleting or popping the array will have no effect over the `mapping` elements. The same happens, for example, if a `mapping` is used as the type of a member field of a `struct` that is the base type of a dynamic storage array. The `mapping` is also ignored in assignments of structs or arrays containing a `mapping`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Map {
    mapping (uint => uint)[] array;

    function allocate(uint newMaps) public {
        for (uint i = 0; i < newMaps; i++)
            array.push();
    }

    function writeMap(uint map, uint key, uint value) public {
        array[map][key] = value;
    }

    function readMap(uint map, uint key) public view returns (uint) {
        return array[map][key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

(continues on next page)

(continued from previous page)

```
    }  
}
```

Consider the example above and the following sequence of calls: `allocate(10), writeMap(4, 128, 256)`. At this point, calling `readMap(4, 128)` returns 256. If we call `eraseMaps`, the length of state variable `array` is zeroed, but since its mapping elements cannot be zeroed, their information stays alive in the contract's storage. After deleting `array`, calling `allocate(5)` allows us to access `array[4]` again, and calling `readMap(4, 128)` returns 256 even without another call to `writeMap`.

If your `mapping` information must be deleted, consider using a library similar to `iterable mapping`, allowing you to traverse the keys and delete their values in the appropriate `mapping`.

Minor Details

- Types that do not occupy the full 32 bytes might contain “dirty higher order bits”. This is especially important if you access `msg.data` - it poses a malleability risk: You can craft transactions that call a function `f(uint8 x)` with a raw byte argument of `0xff000001` and with `0x00000001`. Both are fed to the contract and both will look like the number 1 as far as `x` is concerned, but `msg.data` will be different, so if you use `keccak256(msg.data)` for anything, you will get different results.

3.28.2 Recommendations

Take Warnings Seriously

If the compiler warns you about something, you should change it. Even if you do not think that this particular warning has security implications, there might be another issue buried beneath it. Any compiler warning we issue can be silenced by slight changes to the code.

Always use the latest version of the compiler to be notified about all recently introduced warnings.

Messages of type `info` issued by the compiler are not dangerous, and simply represent extra suggestions and optional information that the compiler thinks might be useful to the user.

Restrict the Amount of Ether

Restrict the amount of Ether (or other tokens) that can be stored in a smart contract. If your source code, the compiler or the platform has a bug, these funds may be lost. If you want to limit your loss, limit the amount of Ether.

Keep it Small and Modular

Keep your contracts small and easily understandable. Single out unrelated functionality in other contracts or into libraries. General recommendations about source code quality of course apply: Limit the amount of local variables, the length of functions and so on. Document your functions so that others can see what your intention was and whether it is different than what the code does.

Use the Checks-Effects-Interactions Pattern

Most functions will first perform some checks (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.). These checks should be done first.

As the second step, if all checks passed, effects to the state variables of the current contract should be made. Interaction with other contracts should be the very last step in any function.

Early contracts delayed some effects and waited for external function calls to return in a non-error state. This is often a serious mistake because of the re-entrancy problem explained above.

Note that, also, calls to known contracts might in turn cause calls to unknown contracts, so it is probably better to just always apply this pattern.

Include a Fail-Safe Mode

While making your system fully decentralised will remove any intermediary, it might be a good idea, especially for new code, to include some kind of fail-safe mechanism:

You can add a function in your smart contract that performs some self-checks like “Has any Ether leaked?”, “Is the sum of the tokens equal to the balance of the contract?” or similar things. Keep in mind that you cannot use too much gas for that, so help through off-chain computations might be needed there.

If the self-check fails, the contract automatically switches into some kind of “failsafe” mode, which, for example, disables most of the features, hands over control to a fixed and trusted third party or just converts the contract into a simple “give me back my money” contract.

Ask for Peer Review

The more people examine a piece of code, the more issues are found. Asking people to review your code also helps as a cross-check to find out whether your code is easy to understand - a very important criterion for good smart contracts.

3.29 SMTChecker and Formal Verification

Using formal verification it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

Solidity implements a formal verification approach based on [SMT \(Satisfiability Modulo Theories\)](#) and [Horn](#) solving. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.

- Out of bounds index access.
- Insufficient funds for a transfer.

All the targets above are automatically checked by default if all engines are enabled, except underflow and overflow for Solidity $\geq 0.8.7$.

The potential warnings that the SMTChecker reports are:

- `<failing property> happens here..` This means that the SMTChecker proved that a certain property fails. A counterexample may be given, however in complex situations it may also not show a counterexample. This result may also be a false positive in certain cases, when the SMT encoding adds abstractions for Solidity code that is either hard or impossible to express.
- `<failing property> might happen here.` This means that the solver could not prove either case within the given timeout. Since the result is unknown, the SMTChecker reports the potential failure for soundness. This may be solved by increasing the query timeout, but the problem might also simply be too hard for the engine to solve.

To enable the SMTChecker, you must select *which engine should run*, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

Note: Prior to Solidity 0.8.4, the default way to enable the SMTChecker was via `pragma experimental SMTChecker;` and only the contracts containing the pragma would be analyzed. That pragma has been deprecated, and although it still enables the SMTChecker for backwards compatibility, it will be removed in Solidity 0.9.0. Note also that now using the pragma even in a single file enables the SMTChecker for all files.

Note: The lack of warnings for a verification target represents an undisputed mathematical proof of correctness, assuming no bugs in the SMTChecker and the underlying solver. Keep in mind that these problems are *very hard* and sometimes *impossible* to solve automatically in the general case. Therefore, several properties might not be solved or might lead to false positives for large contracts. Every proven property should be seen as an important achievement. For advanced users, see *SMTChecker Tuning* to learn a few options that might help proving more complex properties.

3.29.1 Tutorial

Overflow

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }
}
```

(continues on next page)

(continued from previous page)

```
function stateAdd() public view returns (uint) {
    return add(x, y);
}
```

The contract above shows an overflow check example. The SMTChecker does not check underflow and overflow by default for Solidity >=0.8.7, so we need to use the command line option `--model-checker-targets "underflow, overflow"` or the JSON option `settings.modelChecker.targets = ["underflow", "overflow"]`. See [this section for targets configuration](#). Here, it reports the following:

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y = 115792089237316195423570985008687907853269984665640564039457584007913129639935
= 0

Transaction trace:
Overflow.constructor(1,
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935)
State: x = 1, y =
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
    Overflow.add(1,
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935) --_
    ↳ internal call
    --> o.sol:9:20:
    |
9 |         return x_ + y_;
    |         ^^^^^^^^
```

If we add `require` statements that filter out overflow cases, the SMTChecker proves that no overflow is reachable (by not reporting warnings):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}
```

Assert

An assertion represents an invariant in your code: a property that must be true *for all transactions, including all input and storage values*, otherwise there is a bug.

The code below defines a function `f` that guarantees no overflow. Function `inv` defines the specification that `f` is monotonically increasing: for every possible pair (a, b) , if $b > a$ then $f(b) > f(a)$. Since `f` is indeed monotonically increasing, the SMTChecker proves that our property is correct. You are encouraged to play with the property and the function definition to see what results come out!

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Monotonic {
    function f(uint x) internal pure returns (uint) {
        require(x < type(uint128).max);
        return x * 42;
    }

    function inv(uint a, uint b) public pure {
        require(b > a);
        assert(f(b) > f(a));
    }
}
```

We can also add assertions inside loops to verify more complicated properties. The following code searches for the maximum element of an unrestricted array of numbers, and asserts the property that the found element must be greater or equal every element in the array.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m >= a[i]);

        return m;
    }
}
```

Note that in this example the SMTChecker will automatically try to prove three properties:

1. `++i` in the first loop does not overflow.
2. `++i` in the second loop does not overflow.
3. The assertion is always true.

Note: The properties involve loops, which makes it *much much* harder than the previous examples, so beware of loops!

All the properties are correctly proven safe. Feel free to change the properties and/or add restrictions on the array to see different results. For example, changing the code to

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        require(a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m > a[i]);

        return m;
    }
}
```

gives us:

Warning: CHC: Assertion violation happens here.

Counterexample:

```
a = [0, 0, 0, 0, 0]
= 0
```

Transaction trace:

```
Test.constructor()
Test.max([0, 0, 0, 0, 0])
--> max.sol:14:4:
|
14 |         assert(m > a[i]);
```

State Properties

So far the examples only demonstrated the use of the SMTChecker over pure code, proving properties about specific operations or algorithms. A common type of properties in smart contracts are properties that involve the state of the contract. Multiple transactions might be needed to make an assertion fail for such a property.

As an example, consider a 2D grid where both axis have coordinates in the range (- 2^{128} , $2^{128} - 1$). Let us place a robot at position (0, 0). The robot can only move diagonally, one step at a time, and cannot move outside the grid. The robot's state machine can be represented by the smart contract below.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Robot {
    int x = 0;
    int y = 0;
```

(continues on next page)

(continued from previous page)

```

modifier wall {
    require(x > type(int128).min && x < type(int128).max);
    require(y > type(int128).min && y < type(int128).max);
    -;
}

function moveLeftUp() wall public {
    --x;
    ++y;
}

function moveLeftDown() wall public {
    --x;
    --y;
}

function moveRightUp() wall public {
    ++x;
    ++y;
}

function moveRightDown() wall public {
    ++x;
    --y;
}

function inv() public view {
    assert((x + y) % 2 == 0);
}
}

```

Function `inv` represents an invariant of the state machine that `x + y` must be even. The SMTChecker manages to prove that regardless how many commands we give the robot, even if infinitely many, the invariant can *never* fail. The interested reader may want to prove that fact manually as well. Hint: this invariant is inductive.

We can also trick the SMTChecker into giving us a path to a certain position we think might be reachable. We can add the property that (2, 4) is *not* reachable, by adding the following function.

```

function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}

```

This property is false, and while proving that the property is false, the SMTChecker tells us exactly *how* to reach (2, 4):

Warning: CHC: Assertion violation happens here.

Counterexample:

`x = 2, y = 4`

Transaction trace:

`Robot.constructor()`

`State: x = 0, y = 0`

`Robot.moveLeftUp()`

`State: x = (- 1), y = 1`

(continues on next page)

(continued from previous page)

```

Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
--> r.sol:35:4:
|
35 |         assert(!(x == 2 && y == 4));
|         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

Note that the path above is not necessarily deterministic, as there are other paths that could reach (2, 4). The choice of which path is shown might change depending on the used solver, its version, or just randomly.

External Calls and Reentrancy

Every external call is treated as a call to unknown code by the SMTChecker. The reasoning behind that is that even if the code of the called contract is available at compile time, there is no guarantee that the deployed contract will indeed be the same as the contract where the interface came from at compile time.

In some cases, it is possible to automatically infer properties over state variables that are still true even if the externally called code can do anything, including reenter the caller contract.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown u) {
        require(address(u) != address(0));
        unknown = u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint x_) mutex public {
        x = x_;
    }
}

```

(continues on next page)

(continued from previous page)

```
function run() mutex public {
    uint xPre = x;
    unknown.run();
    assert(xPre == x);
}
}
```

The example above shows a contract that uses a mutex flag to forbid reentrancy. The solver is able to infer that when `unknown.run()` is called, the contract is already “locked”, so it would not be possible to change the value of `x`, regardless of what the unknown called code does.

If we “forget” to use the `mutex` modifier on function `set`, the SMTChecker is able to synthesize the behaviour of the externally called code so that the assertion fails:

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
    unknown.run() -- untrusted external call, synthesized as:
        Mutex.set(1) -- reentrant call
--> m.sol:32:3:
|
32 |         assert(xPre == x);
|         ^^^^^^^^^^^^^^^^^^
```

3.29.2 SMTChecker Options and Tuning

Timeout

The SMTChecker uses a hardcoded resource limit (`rlimit`) chosen per solver, which is not precisely related to time. We chose the `rlimit` option as the default because it gives more determinism guarantees than time inside the solver.

This options translates roughly to “a few seconds timeout” per query. Of course many properties are very complex and need a lot of time to be solved, where determinism does not matter. If the SMTChecker does not manage to solve the contract properties with the default `rlimit`, a timeout can be given in milliseconds via the CLI option `--model-checker-timeout <time>` or the JSON option `settings.modelChecker.timeout=<time>`, where 0 means no timeout.

Verification Targets

The types of verification targets created by the SMTChecker can also be customized via the CLI option `--model-checker-target <targets>` or the JSON option `settings.modelChecker.targets=<targets>`. In the CLI case, `<targets>` is a no-space-comma-separated list of one or more verification targets, and an array of one or more targets as strings in the JSON input. The keywords that represent the targets are:

- Assertions: `assert`.
- Arithmetic underflow: `underflow`.
- Arithmetic overflow: `overflow`.
- Division by zero: `divByZero`.
- Trivial conditions and unreachable code: `constantCondition`.
- Popping an empty array: `popEmptyArray`.
- Out of bounds array/fixed bytes index access: `outOfBounds`.
- Insufficient funds for a transfer: `balance`.
- All of the above: `default` (CLI only).

A common subset of targets might be, for example: `--model-checker-targets assert,overflow`.

All targets are checked by default, except underflow and overflow for Solidity $\geq 0.8.7$.

There is no precise heuristic on how and when to split verification targets, but it can be useful especially when dealing with large contracts.

Unproved Targets

If there are any unproved targets, the SMTChecker issues one warning stating how many unproved targets there are. If the user wishes to see all the specific unproved targets, the CLI option `--model-checker-show-unproved` and the JSON option `settings.modelChecker.showUnproved = true` can be used.

Verified Contracts

By default all the deployable contracts in the given sources are analyzed separately as the one that will be deployed. This means that if a contract has many direct and indirect inheritance parents, all of them will be analyzed on their own, even though only the most derived will be accessed directly on the blockchain. This causes an unnecessary burden on the SMTChecker and the solver. To aid cases like this, users can specify which contracts should be analyzed as the deployed one. The parent contracts are of course still analyzed, but only in the context of the most derived contract, reducing the complexity of the encoding and generated queries. Note that abstract contracts are by default not analyzed as the most derived by the SMTChecker.

The chosen contracts can be given via a comma-separated list (whitespace is not allowed) of `<source>:<contract>` pairs in the CLI: `--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>,<source2.sol:contract3>"`, and via the object `settings.modelChecker.contracts` in the *JSON input*, which has the following form:

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

Reported Inferred Inductive Invariants

For properties that were proved safe with the CHC engine, the SMTChecker can retrieve inductive invariants that were inferred by the Horn solver as part of the proof. Currently two types of invariants can be reported to the user:

- Contract Invariants: these are properties over the contract's state variables that are true before and after every possible transaction that the contract may ever run. For example, $x \geq y$, where x and y are a contract's state variables.
- Reentrancy Properties: they represent the behavior of the contract in the presence of external calls to unknown code. These properties can express a relation between the value of the state variables before and after the external call, where the external call is free to do anything, including making reentrant calls to the analyzed contract. Primed variables represent the state variables' values after said external call. Example: $lock \rightarrow x = x'$.

The user can choose the type of invariants to be reported using the CLI option `--model-checker-invariants "contract,reentrancy"` or as an array in the field `settings.modelChecker.invariants` in the [JSON input](#). By default the SMTChecker does not report invariants.

Division and Modulo With Slack Variables

Spacer, the default Horn solver used by the SMTChecker, often dislikes division and modulo operations inside Horn rules. Because of that, by default the Solidity division and modulo operations are encoded using the constraint $a = b * d + m$ where $d = a / b$ and $m = a \% b$. However, other solvers, such as Eldarica, prefer the syntactically precise operations. The command line flag `--model-checker-div-mod-no-slacks` and the JSON option `settings.modelChecker.divModNoSlacks` can be used to toggle the encoding depending on the used solver preferences.

Natspec Function Abstraction

Certain functions including common math methods such as `pow` and `sqrt` may be too complex to be analyzed in a fully automated way. These functions can be annotated with Natspec tags that indicate to the SMTChecker that these functions should be abstracted. This means that the body of the function is not used, and when called, the function will:

- Return a nondeterministic value, and either keep the state variables unchanged if the abstracted function is `view/pure`, or also set the state variables to nondeterministic values otherwise. This can be used via the annotation `/// @custom:smtchecker abstract-function-nondet`.
- Act as an uninterpreted function. This means that the semantics of the function (given by the body) are ignored, and the only property this function has is that given the same input it guarantees the same output. This is currently under development and will be available via the annotation `/// @custom:smtchecker abstract-function-uf`.

Model Checking Engines

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`.

Bounded Model Checker (BMC)

The BMC engine analyzes functions in isolation, that is, it does not take the overall behavior of the contract over multiple transactions into account when analyzing each function. Loops are also ignored in this engine at the moment. Internal function calls are inlined as long as they are not recursive, directly or indirectly. External function calls are inlined if possible. Knowledge that is potentially affected by reentrancy is erased.

The characteristics above make BMC prone to reporting false positives, but it is also lightweight and should be able to quickly find small local bugs.

Constrained Horn Clauses (CHC)

A contract's Control Flow Graph (CFG) is modelled as a system of Horn clauses, where the life cycle of the contract is represented by a loop that can visit every public/external function non-deterministically. This way, the behavior of the entire contract over an unbounded number of transactions is taken into account when analyzing any function. Loops are fully supported by this engine. Internal function calls are supported, and external function calls assume the called code is unknown and can do anything.

The CHC engine is much more powerful than BMC in terms of what it can prove, and might require more computing resources.

SMT and Horn solvers

The two engines detailed above use automated theorem provers as their logical backends. BMC uses an SMT solver, whereas CHC uses a Horn solver. Often the same tool can act as both, as seen in [z3](#), which is primarily an SMT solver and makes [Spacer](#) available as a Horn solver, and [Eldarica](#) which does both.

The user can choose which solvers should be used, if available, via the CLI option `--model-checker-solvers {all,cvc4,smtlib2,z3}` or the JSON option `settings.modelChecker.solvers=[smtlib2,z3]`, where:

- `cvc4` is only available if the `solc` binary is compiled with it. Only BMC uses `cvc4`.
- `smtlib2` outputs SMT/Horn queries in the `smtlib2` format. These can be used together with the compiler's [callback mechanism](#) so that any solver binary from the system can be employed to synchronously return the results of the queries to the compiler. This is currently the only way to use Eldarica, for example, since it does not have a C++ API. This can be used by both BMC and CHC depending on which solvers are called.
- `z3` is available
 - if `solc` is compiled with it;
 - if a dynamic `z3` library of version 4.8.x is installed in a Linux system (from Solidity 0.7.6);
 - statically in `soljson.js` (from Solidity 0.6.9), that is, the Javascript binary of the compiler.

Note: `z3` version 4.8.16 broke ABI compatibility with previous versions and cannot be used with `solc <=0.8.13`. If you are using `z3 >=4.8.16` please use `solc >=0.8.14`.

Since both BMC and CHC use `z3`, and `z3` is available in a greater variety of environments, including in the browser, most users will almost never need to be concerned about this option. More advanced users might apply this option to try alternative solvers on more complex problems.

Please note that certain combinations of chosen engine and solver will lead to the SMTChecker doing nothing, for example choosing CHC and `cvc4`.

3.29.3 Abstraction and False Positives

The SMTChecker implements abstractions in an incomplete and sound way: If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type). If it determines that a verification target is safe, it is indeed safe, that is, there are no false negatives (unless there is a bug in the SMTChecker).

If a target cannot be proven you can try to help the solver by using the tuning options in the previous section. If you are sure of a false positive, adding `require` statements in the code with more information may also give some more power to the solver.

SMT Encoding and Types

The SMTChecker encoding tries to be as precise as possible, mapping Solidity types and expressions to their closest [SMT-LIB](#) representation, as shown in the table below.

Solidity type	SMT sort	Theories
Boolean	Bool	Bool
intN, uintN, address, bytesN, enum, contract	Integer	LIA, NIA
array, mapping, bytes, string	Tuple (Array elements, Integer length)	Datatypes, Arrays, LIA
struct	Tuple	Datatypes
other types	Integer	LIA

Types that are not yet supported are abstracted by a single 256-bit unsigned integer, where their unsupported operations are ignored.

For more details on how the SMT encoding works internally, see the paper [SMT-based Verification of Solidity Smart Contracts](#).

Function Calls

In the BMC engine, function calls to the same contract (or base contracts) are inlined when possible, that is, when their implementation is available. Calls to functions in other contracts are not inlined even if their code is available, since we cannot guarantee that the actual deployed code is the same.

The CHC engine creates nonlinear Horn clauses that use summaries of the called functions to support internal function calls. External function calls are treated as calls to unknown code, including potential reentrant calls.

Complex pure functions are abstracted by an uninterpreted function (UF) over the arguments.

Functions	BMC/CHC behavior
<code>assert</code>	Verification target.
<code>require</code>	Assumption.
<code>internal call</code>	BMC: Inline function call. CHC: Function summaries.
<code>external call to known code</code>	BMC: Inline function call or erase knowledge about state variables and local storage references. CHC: Assume called code is unknown. Try to infer invariants that hold after the call returns.
<code>Storage array push/pop</code>	Supported precisely. Checks whether it is popping an empty array.
<code>ABI functions</code>	Abstracted with UF.
<code>addmod, mulmod</code>	Supported precisely.
<code>gasleft, blockhash, keccak256, ecrecover ripemd160</code>	Abstracted with UF.
<code>pure functions without implementation (external or complex)</code>	Abstracted with UF
<code>external functions without implementation</code>	BMC: Erase state knowledge and assume result is nondeterministic. CHC: Nondeterministic summary. Try to infer invariants that hold after the call returns.
<code>transfer</code>	BMC: Checks whether the contract's balance is sufficient. CHC: does not yet perform the check.
<code>others</code>	Currently unsupported

Using abstraction means loss of precise knowledge, but in many cases it does not mean loss of proving power.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Recover {
    function f(
        bytes32 hash,
        uint8 v1, uint8 v2,
        bytes32 r1, bytes32 r2,
        bytes32 s1, bytes32 s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, v1, r1, s1);
        require(v1 == v2);
        require(r1 == r2);
        require(s1 == s2);
        address a2 = ecrecover(hash, v2, r2, s2);
        assert(a1 == a2);
        return a1;
    }
}
```

In the example above, the SMTChecker is not expressive enough to actually compute `ecrecover`, but by modelling the function calls as uninterpreted functions we know that the return value is the same when called on equivalent parameters. This is enough to prove that the assertion above is always true.

Abstracting a function call with an UF can be done for functions known to be deterministic, and can be easily done for pure functions. It is however difficult to do this with general external functions, since they might depend on state variables.

Reference Types and Aliasing

Solidity implements aliasing for reference types with the same *data location*. That means one variable may be modified through a reference to the same data area. The SMTChecker does not keep track of which references refer to the same data. This implies that whenever a local reference or state variable of reference type is assigned, all knowledge regarding variables of the same type and data location is erased. If the type is nested, the knowledge removal also includes all the prefix base types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
        // Erasing knowledge about memory references should not
        // erase knowledge about state variables.
        assert(array1[0] == 42);
        // However, an assignment to a storage reference will erase
        // storage knowledge accordingly.
        d[0] = 2;
        // Fails as false positive because of the assignment above.
        assert(array1[0] == 42);
        // Fails because `a == b` is possible.
        assert(a[0] == 2);
        // Fails because `c[i] == b` is possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
        f(a, b, c, array2[x]);
    }
}
```

After the assignment to `b[0]`, we need to clear knowledge about `a` since it has the same type (`uint[]`) and data location (memory). We also need to clear knowledge about `c`, since its base type is also a `uint[]` located in memory. This implies that some `c[i]` could refer to the same data as `b` or `a`.

Notice that we do not clear knowledge about `array` and `d` because they are located in storage, even though they also

have type `uint[]`. However, if `d` was assigned, we would need to clear knowledge about `array` and vice-versa.

Contract Balance

A contract may be deployed with funds sent to it, if `msg.value > 0` in the deployment transaction. However, the contract's address may already have funds before deployment, which are kept by the contract. Therefore, the SMTChecker assumes that `address(this).balance >= msg.value` in the constructor in order to be consistent with the EVM rules. The contract's balance may also increase without triggering any calls to the contract, if

- `selfdestruct` is executed by another contract with the analyzed contract as the target of the remaining funds,
- the contract is the coinbase (i.e., `block.coinbase`) of some block.

To model this properly, the SMTChecker assumes that at every new transaction the contract's balance may grow by at least `msg.value`.

3.29.4 Real World Assumptions

Some scenarios can be expressed in Solidity and the EVM, but are expected to never occur in practice. One of such cases is the length of a dynamic storage array overflowing during a push: If the push operation is applied to an array of length $2^{256} - 1$, its length silently overflows. However, this is unlikely to happen in practice, since the operations required to grow the array to that point would take billions of years to execute. Another similar assumption taken by the SMTChecker is that an address' balance can never overflow.

A similar idea was presented in [EIP-1985](#).

3.30 Resources

3.30.1 General Resources

- Ethereum.org Developer Portal
- Ethereum StackExchange
- Solidity Portal
- Solidity Changelog
- Solidity Source Code on GitHub
- Solidity Language Users Chat
- Solidity Compiler Developers Chat
- Awesome Solidity
- Solidity by Example
- Solidity Documentation Community Translations

3.30.2 Integrated (Ethereum) Development Environments

- **Brownie**
Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.
- **Dapp**
Tool for building, testing and deploying smart contracts from the command line.
- **Embark**
Developer platform for building and deploying decentralized applications.
- **Foundry**
Fast, portable and modular toolkit for Ethereum application development written in Rust.
- **Hardhat**
Ethereum development environment with local Ethereum network, debugging features and plugin ecosystem.
- **Remix**
Browser-based IDE with integrated compiler and Solidity runtime environment without server-side components.
- **Truffle**
Ethereum development framework.

3.30.3 Editor Integrations

- Atom
 - **Etheratom**
Plugin for the Atom editor that features syntax highlighting, compilation and a runtime environment (Backend node & VM compatible).
 - **Atom Solidity Linter**
Plugin for the Atom editor that provides Solidity linting.
 - **Atom Solium Linter**
Configurable Solidity linter for Atom using Solium (now Ethlint) as a base.
- Emacs
 - **Emacs Solidity**
Plugin for the Emacs editor providing syntax highlighting and compilation error reporting.
- IntelliJ
 - **IntelliJ IDEA plugin**
Solidity plugin for IntelliJ IDEA (and all other JetBrains IDEs)
- Sublime
 - **Package for SublimeText - Solidity language syntax**
Solidity syntax highlighting for SublimeText editor.
- Vim
 - **Vim Solidity**
Plugin for the Vim editor providing syntax highlighting.
 - **Vim Syntastic**
Plugin for the Vim editor providing compile checking.

- Visual Studio Code
 - **Visual Studio Code extension**
Solidity plugin for Microsoft Visual Studio Code that includes syntax highlighting and the Solidity compiler.
 - **Solidity Visual Auditor extension**
Adds security centric syntax and semantic highlighting to Visual Studio Code.

3.30.4 Solidity Tools

- **ABI to Solidity interface converter**
A script for generating contract interfaces from the ABI of a smart contract.
- **abi-to-sol**
Tool to generate Solidity interface source from a given ABI JSON.
- **Doxity**
Documentation Generator for Solidity.
- **Ethlint**
Linter to identify and fix style and security issues in Solidity.
- **evmdis**
EVM Disassembler that performs static analysis on the bytecode to provide a higher level of abstraction than raw EVM operations.
- **EVM Lab**
Rich tool package to interact with the EVM. Includes a VM, Etherchain API, and a trace-viewer with gas cost display.
- **hevm**
EVM debugger and symbolic execution engine.
- **leaflet**
A documentation generator for Solidity smart-contracts.
- **PIET**
A tool to develop, audit and use Solidity smart contracts through a simple graphical interface.
- **Scaffold-ETH**
Forkable Ethereum development stack focused on fast product iterations.
- **sol2uml**
Unified Modeling Language (UML) class diagram generator for Solidity contracts.
- **solf-select**
A script to quickly switch between Solidity compiler versions.
- **Solidity prettier plugin**
A Prettier Plugin for Solidity.
- **Solidity REPL**
Try Solidity instantly with a command-line Solidity console.
- **solgraph**
Visualize Solidity control flow and highlight potential security vulnerabilities.
- **Solhint**
Solidity linter that provides security, style guide and best practice rules for smart contract validation.

- **Sourcify**

Decentralized automated contract verification service and public repository of contract metadata.

- **Sūrya**

Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts' structure. Also supports querying the function call graph.

- **Universal Mutator**

A tool for mutation generation, with configurable rules and support for Solidity and Vyper.

3.30.5 Third-Party Solidity Parsers and Grammars

- **Solidity Parser for JavaScript**

A Solidity parser for JS built on top of a robust ANTLR4 grammar.

3.31 Import Path Resolution

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. Paths used in imports must work the same way everywhere while the command-line interface must be able to work with platform-specific paths to provide good user experience. This section aims to explain in detail how Solidity reconciles these requirements.

3.31.1 Virtual Filesystem

The compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. When you use the [import statement](#), you specify an *import path* that references a source unit name.

Import Callback

The VFS is initially populated only with files the compiler has received as input. Additional files can be loaded during compilation using an *import callback*, which is different depending on the type of compiler you use (see below). If the compiler does not find any source unit name matching the import path in the VFS, it invokes the callback, which is responsible for obtaining the source code to be placed under that name. An import callback is free to interpret source unit names in an arbitrary way, not just as paths. If there is no callback available when one is needed or if it fails to locate the source code, compilation fails.

The command-line compiler provides the *Host Filesystem Loader* - a rudimentary callback that interprets a source unit name as a path in the local filesystem. The [JavaScript interface](#) does not provide any by default, but one can be provided by the user. This mechanism can be used to obtain source code from locations other than the local filesystem (which may not even be accessible, e.g. when the compiler is running in a browser). For example the [Remix IDE](#) provides a versatile callback that lets you [import files from HTTP, IPFS and Swarm URLs](#) or refer directly to packages in [NPM registry](#).

Note: Host Filesystem Loader's file lookup is platform-dependent. For example backslashes in a source unit name can be interpreted as directory separators or not and the lookup can be case-sensitive or not, depending on the underlying platform.

For portability it is recommended to avoid using import paths that will work correctly only with a specific import callback or only on one platform. For example you should always use forward slashes since they work as path separators also on platforms that support backslashes.

Initial Content of the Virtual Filesystem

The initial content of the VFS depends on how you invoke the compiler:

1. `solc` / command-line interface

When you compile a file using the command-line interface of the compiler, you provide one or more paths to files containing Solidity code:

```
solc contract.sol /usr/local/dapp-bin/token.sol
```

The source unit name of a file loaded this way is constructed by converting its path to a canonical form and, if possible, making it relative to either the base path or one of the include paths. See [CLI Path Normalization and Stripping](#) for a detailed description of this process.

2. Standard JSON

When using the [Standard JSON](#) API (via either the JavaScript interface or the `--standard-json` command-line option) you provide input in JSON format, containing, among other things, the content of all your source files:

```
{
  "language": "Solidity",
  "sources": {
    "contract.sol": {
      "content": "import \"../util.sol\";\ncontract C {}"
    },
    "util.sol": {
      "content": "library Util {}"
    },
    "/usr/local/dapp-bin/token.sol": {
      "content": "contract Token {}"
    }
  },
  "settings": {"outputSelection": {"*": { "*": ["metadata", "evm bytecode"]}}}
}
```

The sources dictionary becomes the initial content of the virtual filesystem and its keys are used as source unit names.

3. Standard JSON (via import callback)

With Standard JSON it is also possible to tell the compiler to use the import callback to obtain the source code:

```
{
  "language": "Solidity",
  "sources": {
    "/usr/local/dapp-bin/token.sol": {
      "urls": [
        "/projects/mytoken.sol",
        "https://example.com/projects/mytoken.sol"
      ]
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

        },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm bytecode"]}}}
}
```

If an import callback is available, the compiler will give it the strings specified in `urls` one by one, until one is loaded successfully or the end of the list is reached.

The source unit names are determined the same way as when using `content` - they are keys of the `sources` dictionary and the content of `urls` does not affect them in any way.

4. Standard input

On the command line it is also possible to provide the source by sending it to compiler's standard input:

```
echo 'import "./util.sol"; contract C {}' | solc -
```

- used as one of the arguments instructs the compiler to place the content of the standard input in the virtual filesystem under a special source unit name: `<stdin>`.

Once the VFS is initialized, additional files can still be added to it only through the import callback.

3.31.2 Imports

The import statement specifies an *import path*. Based on how the import path is specified, we can divide imports into two categories:

- *Direct imports*, where you specify the full source unit name directly.
- *Relative imports*, where you specify a path starting with `./` or `../` to be combined with the source unit name of the importing file.

Listing 1: contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

In the above `./math/math.sol` and `contracts/tokens/token.sol` are import paths while the source unit names they translate to are `contracts/math/math.sol` and `contracts/tokens/token.sol` respectively.

Direct Imports

An import that does not start with `./` or `../` is a *direct import*.

```
import "/project/lib/util.sol";           // source unit name: /project/lib/util.sol
import "lib/util.sol";                   // source unit name: lib/util.sol
import "@openzeppelin/address.sol";       // source unit name: @openzeppelin/address.sol
import "https://example.com/token.sol";  // source unit name: https://example.com/token.
                                         ↵sol
```

After applying any *import remappings* the import path simply becomes the source unit name.

Note: A source unit name is just an identifier and even if its value happens to look like a path, it is not subject to the normalization rules you would typically expect in a shell. Any `./` or `../` segments or sequences of multiple

slashes remain a part of it. When the source is provided via Standard JSON interface it is entirely possible to associate different content with source unit names that would refer to the same file on disk.

When the source is not available in the virtual filesystem, the compiler passes the source unit name to the import callback. The Host Filesystem Loader will attempt to use it as a path and look up the file on disk. At this point the platform-specific normalization rules kick in and names that were considered different in the VFS may actually result in the same file being loaded. For example `/project/lib/math.sol` and `/project/lib/../lib///math.sol` are considered completely different in the VFS even though they refer to the same file on disk.

Note: Even if an import callback ends up loading source code for two different source unit names from the same file on disk, the compiler will still see them as separate source units. It is the source unit name that matters, not the physical location of the code.

Relative Imports

An import starting with `.` or `..` is a *relative import*. Such imports specify a path relative to the source unit name of the importing source unit:

Listing 2: `/project/lib/math.sol`

```
import "./util.sol" as util;    // source unit name: /project/lib/util.sol
import "../token.sol" as token; // source unit name: /project/token.sol
```

Listing 3: `lib/math.sol`

```
import "./util.sol" as util;    // source unit name: lib/util.sol
import "../token.sol" as token; // source unit name: token.sol
```

Note: Relative imports **always** start with `.` or `..` so `import "util.sol"`, unlike `import "./util.sol"`, is a direct import. While both paths would be considered relative in the host filesystem, `util.sol` is actually absolute in the VFS.

Let us define a *path segment* as any non-empty part of the path that does not contain a separator and is bounded by two path separators. A separator is a forward slash or the beginning/end of the string. For example in `./abc/.../` there are three path segments: `..`, `abc` and `...`

The compiler computes a source unit name from the import path in the following way:

1. First a prefix is computed
 - Prefix is initialized with the source unit name of the importing source unit.
 - The last path segment with preceding slashes is removed from the prefix.
 - Then, the leading part of the normalized import path, consisting only of `/` and `.` characters is considered. For every `..` segment found in this part the last path segment with preceding slashes is removed from the prefix.
2. Then the prefix is prepended to the normalized import path. If the prefix is non-empty, a single slash is inserted between it and the import path.

The removal of the last path segment with preceding slashes is understood to work as follows:

1. Everything past the last slash is removed (i.e. `a/b//c.sol` becomes `a/b//`).

2. All trailing slashes are removed (i.e. a/b// becomes a/b).

The normalization rules are the same as for UNIX paths, namely:

- All the internal . segments are removed.
- Every internal .. segment backtracks one level up in the hierarchy.
- Multiple slashes are squashed into a single one.

Note that normalization is performed only on the import path. The source unit name of the importing module that is used for the prefix remains unnormalized. This ensures that the `protocol://` part does not turn into `protocol:/` if the importing file is identified with a URL.

If your import paths are already normalized, you can expect the above algorithm to produce very intuitive results. Here are some examples of what you can expect if they are not:

Listing 4: lib/src/../contract.sol

```
import "./util./util.sol";           // source unit name: lib/src/../util/util.sol
import "./util//util.sol";          // source unit name: lib/src/../util/util.sol
import "../util///array/util.sol";   // source unit name: lib/src/array/util.sol
import "../../.../util.sol";        // source unit name: util.sol
import "../../../../util.sol";       // source unit name: util.sol
```

Note: The use of relative imports containing leading .. segments is not recommended. The same effect can be achieved in a more reliable way by using direct imports with *base path and include paths*.

3.31.3 Base Path and Include Paths

The base path and include paths represent directories that the Host Filesystem Loader will load files from. When a source unit name is passed to the loader, it prepends the base path to it and performs a filesystem lookup. If the lookup does not succeed, the same is done with all directories on the include path list.

It is recommended to set the base path to the root directory of your project and use include paths to specify additional locations that may contain libraries your project depends on. This lets you import from these libraries in a uniform way, no matter where they are located in the filesystem relative to your project. For example, if you use npm to install packages and your contract imports `@openzeppelin/contracts/utils/Strings.sol`, you can use these options to tell the compiler that the library can be found in one of the npm package directories:

```
solc contract.sol \
  --base-path . \
  --include-path node_modules/ \
  --include-path /usr/local/lib/node_modules/
```

Your contract will compile (with the same exact metadata) no matter whether you install the library in the local or global package directory or even directly under your project root.

By default the base path is empty, which leaves the source unit name unchanged. When the source unit name is a relative path, this results in the file being looked up in the directory the compiler has been invoked from. It is also the only value that results in absolute paths in source unit names being actually interpreted as absolute paths on disk. If the base path itself is relative, it is interpreted as relative to the current working directory of the compiler.

Note: Include paths cannot have empty values and must be used together with a non-empty base path.

Note: Include paths and base path can overlap as long as it does not make import resolution ambiguous. For example, you can specify a directory inside base path as an include directory or have an include directory that is a subdirectory of another include directory. The compiler will only issue an error if the source unit name passed to the Host Filesystem Loader represents an existing path when combined with multiple include paths or an include path and base path.

CLI Path Normalization and Stripping

On the command line the compiler behaves just as you would expect from any other program: it accepts paths in a format native to the platform and relative paths are relative to the current working directory. The source unit names assigned to files whose paths are specified on the command line, however, should not change just because the project is being compiled on a different platform or because the compiler happens to have been invoked from a different directory. To achieve this, paths to source files coming from the command line must be converted to a canonical form, and, if possible, made relative to the base path or one of the include paths.

The normalization rules are as follows:

- If a path is relative, it is made absolute by prepending the current working directory to it.
- Internal . and .. segments are collapsed.
- Platform-specific path separators are replaced with forward slashes.
- Sequences of multiple consecutive path separators are squashed into a single separator (unless they are the leading slashes of an [UNC path](#)).
- If the path includes a root name (e.g. a drive letter on Windows) and the root is the same as the root of the current working directory, the root is replaced with /.
- Symbolic links in the path are **not** resolved.
 - The only exception is the path to the current working directory prepended to relative paths in the process of making them absolute. On some platforms the working directory is reported always with symbolic links resolved so for consistency the compiler resolves them everywhere.
- The original case of the path is preserved even if the filesystem is case-insensitive but [case-preserving](#) and the actual case on disk is different.

Note: There are situations where paths cannot be made platform-independent. For example on Windows the compiler can avoid using drive letters by referring to the root directory of the current drive as / but drive letters are still necessary for paths leading to other drives. You can avoid such situations by ensuring that all the files are available within a single directory tree on the same drive.

After normalization the compiler attempts to make the source file path relative. It tries the base path first and then the include paths in the order they were given. If the base path is empty or not specified, it is treated as if it was equal to the path to the current working directory (with all symbolic links resolved). The result is accepted only if the normalized directory path is the exact prefix of the normalized file path. Otherwise the file path remains absolute. This makes the conversion unambiguous and ensures that the relative path does not start with ../. The resulting file path becomes the source unit name.

Note: The relative path produced by stripping must remain unique within the base path and include paths. For example the compiler will issue an error for the following command if both /project/contract.sol and /lib/contract.sol exist:

```
solc /project/contract.sol --base-path /project --include-path /lib
```

Note: Prior to version 0.8.8, CLI path stripping was not performed and the only normalization applied was the conversion of path separators. When working with older versions of the compiler it is recommended to invoke the compiler from the base path and to only use relative paths on the command line.

3.31.4 Allowed Paths

As a security measure, the Host Filesystem Loader will refuse to load files from outside of a few locations that are considered safe by default:

- Outside of Standard JSON mode:
 - The directories containing input files listed on the command line.
 - The directories used as *remapping* targets. If the target is not a directory (i.e does not end with `/`, `./` or `/..`) the directory containing the target is used instead.
 - Base path and include paths.
- In Standard JSON mode:
 - Base path and include paths.

Additional directories can be whitelisted using the `--allow-paths` option. The option accepts a comma-separated list of paths:

```
cd /home/user/project/
solc token/contract.sol \
  lib/util.sol=libs/util.sol \
  --base-path=token/ \
  --include-path=/lib/ \
  --allow-paths=../utils/,/tmp/libraries
```

When the compiler is invoked with the command shown above, the Host Filesystem Loader will allow importing files from the following directories:

- `/home/user/project/token/` (because `token/` contains the input file and also because it is the base path),
- `/lib/` (because `/lib/` is one of the include paths),
- `/home/user/project/libs/` (because `libs/` is a directory containing a remapping target),
- `/home/user/utils/` (because of `../utils/` passed to `--allow-paths`),
- `/tmp/libraries/` (because of `/tmp/libraries` passed to `--allow-paths`),

Note: The working directory of the compiler is one of the paths allowed by default only if it happens to be the base path (or the base path is not specified or has an empty value).

Note: The compiler does not check if allowed paths actually exist and whether they are directories. Non-existent or empty paths are simply ignored. If an allowed path matches a file rather than a directory, the file is considered whitelisted, too.

Note: Allowed paths are case-sensitive even if the filesystem is not. The case must exactly match the one used in your imports. For example `--allow-paths tokens` will not match `import "Tokens/IERC20.sol"`.

Warning: Files and directories only reachable through symbolic links from allowed directories are not automatically whitelisted. For example if `token/contract.sol` in the example above was actually a symlink pointing at `/etc/passwd` the compiler would refuse to load it unless `/etc/` was one of the allowed paths too.

3.31.5 Import Remapping

Import remapping allows you to redirect imports to a different location in the virtual filesystem. The mechanism works by changing the translation between import paths and source unit names. For example you can set up a remapping so that any import from the virtual directory `github.com/ethereum/dapp-bin/library/` would be seen as an import from `dapp-bin/library/` instead.

You can limit the scope of a remapping by specifying a *context*. This allows creating remappings that apply only to imports located in a specific library or a specific file. Without a context a remapping is applied to every matching import in all the files in the virtual filesystem.

Import remappings have the form of `context:prefix=target`:

- `context` must match the beginning of the source unit name of the file containing the import.
- `prefix` must match the beginning of the source unit name resulting from the import.
- `target` is the value the prefix is replaced with.

For example, if you clone <https://github.com/ethereum/dapp-bin/> locally to `/project/dapp-bin` and run the compiler with:

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/
→library/math.sol
```

The compiler will look for the file in the VFS under `dapp-bin/library/math.sol`. If the file is not available there, the source unit name will be passed to the Host Filesystem Loader, which will then look in `/project/dapp-bin/library/iterable_mapping.sol`.

Warning: Information about remappings is stored in contract metadata. Since the binary produced by the compiler has a hash of the metadata embedded in it, any modification to the remappings will result in different bytecode.

For this reason you should be careful not to include any local information in remapping targets. For example if your library is located in `/home/user/packages/mymath/math.sol`, a remapping like `@math=/home/user/packages/mymath/` would result in your home directory being included in the metadata. To be able to reproduce the same bytecode with such a remapping on a different machine, you would need to recreate parts of your local directory structure in the VFS and (if you rely on Host Filesystem Loader) also in the host filesystem.

To avoid having your local directory structure embedded in the metadata, it is recommended to designate the directories containing libraries as *include paths* instead. For example, in the example above `--include-path /home/user/packages/` would let you use imports starting with `mymath/`. Unlike remapping, the option on its own will

not make `mymath` appear as `@math` but this can be achieved by creating a symbolic link or renaming the package subdirectory.

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to `/project/dapp-bin_old`, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
  module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
  --base-path /project \
  source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

Here are the detailed rules governing the behaviour of remappings:

- Remappings only affect the translation between import paths and source unit names.**

Source unit names added to the VFS in any other way cannot be remapped. For example the paths you specify on the command-line and the ones in `sources.urls` in Standard JSON are not affected.

```
solc /project=/contracts/ /project/contract.sol # source unit name: /project/
  ↳contract.sol
```

In the example above the compiler will load the source code from `/project/contract.sol` and place it under that exact source unit name in the VFS, not under `/contract/contract.sol`.

- Context and prefix must match source unit names, not import paths.**

- This means that you cannot remap `./` or `../` directly since they are replaced during the translation to source unit name but you can remap the part of the name they are replaced with:

```
solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
  ↳contract.sol
```

Listing 5: `/project/contract.sol`

```
import "./util.sol" as util; // source unit name: b/util.sol
```

- You cannot remap base path or any other part of the path that is only added internally by an import callback:

```
solc /project=/contracts/ /project/contract.sol --base-path /project # source_
  ↳unit name: contract.sol
```

Listing 6: `/project/contract.sol`

```
import "util.sol" as util; // source unit name: util.sol
```

- Target is inserted directly into the source unit name and does not necessarily have to be a valid path.**

- It can be anything as long as the import callback can handle it. In case of the Host Filesystem Loader this includes also relative paths. When using the JavaScript interface you can even use URLs and abstract identifiers if your callback can handle them.
- Remapping happens after relative imports have already been resolved into source unit names. This means that targets starting with `./` and `../` have no special meaning and are relative to the base path rather than to the location of the source file.

- Remapping targets are not normalized so `@root/=./a/b//` will remap `@root/contract.sol` to `./a/b//contract.sol` and not `a/b/contract.sol`.
- If the target does not end with a slash, the compiler will not add one automatically:

```
solc /project=/contracts /project/contract.sol # source unit name: /project/
      ↳contract.sol
```

Listing 7: /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. Context and prefix are patterns and matches must be exact.

- `a//b=c` will not match `a/b`.
- source unit names are not normalized so `a/b=c` will not match `a//b` either.
- Parts of file and directory names can match as well. `/newProject/con:/new=old` will match `/newProject/contract.sol` and remap it to `oldProject/contract.sol`.

5. At most one remapping is applied to a single import.

- If multiple remappings match the same source unit name, the one with the longest matching prefix is chosen.
- If prefixes are identical, the one specified last wins.
- Remappings do not work on other remappings. For example `a=b b=c c=d` will not result in `a` being remapped to `d`.

6. Prefix cannot be empty but context and target are optional.

- If `target` is the empty string, `prefix` is simply removed from import paths.
- Empty `context` means that the remapping applies to all imports in all source units.

3.31.6 Using URLs in imports

Most URL prefixes such as `https://` or `data://` have no special meaning in import paths. The only exception is `file://` which is stripped from source unit names by the Host Filesystem Loader.

When compiling locally you can use import remapping to replace the protocol and domain part with a local path:

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Note the leading `:`, which is necessary when the remapping context is empty. Otherwise the `https:` part would be interpreted by the compiler as the context.

3.32 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can be compiled to bytecode for different backends.

Support for EVM 1.0, EVM 1.5 and Ewasm is planned, and it is designed to be a usable common denominator of all three platforms. It can already be used in stand-alone mode and for “inline assembly” inside Solidity and there is an experimental implementation of the Solidity compiler that uses Yul as an intermediate language. Yul is a good target for high-level optimisation stages that can benefit all target platforms equally.

3.32.1 Motivation and High-level Description

The design of Yul tries to achieve several goals:

1. Programs written in Yul should be readable, even if the code is generated by a compiler from Solidity or another high-level language.
2. Control flow should be easy to understand to help in manual inspection, formal verification and optimization.
3. The translation from Yul to bytecode should be as straightforward as possible.
4. Yul should be suitable for whole-program optimization.

In order to achieve the first and second goal, Yul provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. These should be sufficient for adequately representing the control flow for assembly programs. Therefore, no explicit statements for `SWAP`, `DUP`, `JUMPDEST`, `JUMP` and `JUMPI` are provided, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

Even though it was designed for stack machines, Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack.

The third goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...) and cleanup of local variables from the stack.

To avoid confusions between concepts like values and references, Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability.

To keep the language simple and flexible, Yul does not have any built-in operations, functions or types in its pure form. These are added together with their semantics when specifying a dialect of Yul, which allows specializing Yul to the requirements of different target platforms and feature sets.

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as builtin functions (see below) and defines only the type `u256`, which is the native 256-bit type of the EVM. Because of that, we will not provide types in the examples below.

3.32.2 Simple Example

The following example program is written in the EVM dialect and computes exponentiation. It can be compiled using `solc --strict-assembly`. The builtin functions `mul` and `div` compute product and division, respectively.

```
{  
    function power(base, exponent) -> result  
    {  
        switch exponent  
        case 0 { result := 1 }  
        case 1 { result := base }  
        default  
        {  
            result := power(mul(base, base), div(exponent, 2))  
            switch mod(exponent, 2)  
                case 1 { result := mul(base, result) }  
        }  
    }  
}
```

It is also possible to implement the same function using a for-loop instead of with recursion. Here, `lt(a, b)` computes whether `a` is less than `b`. less-than comparison.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

At the *end of the section*, a complete implementation of the ERC-20 standard can be found.

3.32.3 Stand-Alone Usage

You can use Yul in its stand-alone form in the EVM dialect using the Solidity compiler. This will use the *Yul object notation* so that it is possible to refer to code as data to deploy contracts. This Yul mode is available for the commandline compiler (use `--strict-assembly`) and for the *standard-json interface*:

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": [ "*" ], "" : [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

Warning: Yul is in active development and bytecode generation is only fully implemented for the EVM dialect of Yul with EVM 1.0 as target.

3.32.4 Informal Description of Yul

In the following, we will talk about each individual aspect of the Yul language. In examples, we will use the default EVM dialect.

Syntax

Yul parses comments, literals and identifiers in the same way as Solidity, so you can e.g. use `//` and `/* */` to denote comments. There is one exception: Identifiers in Yul can contain dots: `..`

Yul can specify “objects” that consist of code, data and sub-objects. Please see *Yul Objects* below for details on that. In this section, we are only concerned with the code part of such an object. This code part always consists of a curly-braces delimited block. Most tools support specifying just a code block where an object is expected.

Inside a code block, the following elements can be used (see the later sections for more details):

- literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)

- calls to builtin functions, e.g. `add(1, mload(0))`
- variable declarations, e.g. `let x := 7, let x := add(y, 3)` or `let x` (initial value of 0 is assigned)
- identifiers (variables), e.g. `add(3, x)`
- assignments, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`
- if statements, e.g. `if lt(a, b) { sstore(0, 1) }`
- switch statements, e.g. `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for loops, e.g. `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- function definitions, e.g. `function f(a, b) -> c { c := add(a, b) }`

Multiple syntactical elements can follow each other simply separated by whitespace, i.e. there is no terminating ; or newline required.

Literals

As literals, you can use:

- Integer constants in decimal or hexadecimal notation.
- ASCII strings (e.g. "abc"), which may contain hex escapes \xNN and Unicode escapes \uNNNN where N are hexadecimal digits.
- Hex strings (e.g. hex"616263").

In the EVM dialect of Yul, literals represent 256-bit words as follows:

- Decimal or hexadecimal constants must be less than $2^{**}256$. They represent the 256-bit word with that value as an unsigned integer in big endian encoding.
- An ASCII string is first viewed as a byte sequence, by viewing a non-escape ASCII character as a single byte whose value is the ASCII code, an escape \xNN as single byte with that value, and an escape \uNNNN as the UTF-8 sequence of bytes for that code point. The byte sequence must not exceed 32 bytes. The byte sequence is padded with zeros on the right to reach 32 bytes in length; in other words, the string is stored left-aligned. The padded byte sequence represents a 256-bit word whose most significant 8 bits are the ones from the first byte, i.e. the bytes are interpreted in big endian form.
- A hex string is first viewed as a byte sequence, by viewing each pair of contiguous hex digits as a byte. The byte sequence must not exceed 32 bytes (i.e. 64 hex digits), and is treated as above.

When compiling for the EVM, this will be translated into an appropriate PUSHi instruction. In the following example, 3 and 2 are added resulting in 5 and then the bitwise and with the string "abc" is computed. The final value is assigned to a local variable called x.

The 32-byte limit above does not apply to string literals passed to builtin functions that require literal arguments (e.g. `setimmutable` or `loadimmutable`). Those strings never end up in the generated bytecode.

```
let x := and("abc", add(3, 2))
```

Unless it is the default type, the type of a literal has to be specified after a colon:

```
// This will not compile (u32 and u256 type not implemented yet)
let x := and("abc":u32, add(3:u256, 2:u256))
```

Function Calls

Both built-in and user-defined functions (see below) can be called in the same way as shown in the previous example. If the function returns a single value, it can be directly used inside an expression again. If it returns multiple values, they have to be assigned to local variables.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Here, the user-defined function `f` returns two values.
let x, y := f(1, mload(0))
```

For built-in functions of the EVM, functional expressions can be directly translated to a stream of opcodes: You just read the expression from right to left to obtain the opcodes. In the case of the first line in the example, this is PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE.

For calls to user-defined functions, the arguments are also put on the stack from right to left and this is the order in which argument lists are evaluated. The return values, though, are expected on the stack from left to right, i.e. in this example, y is on top of the stack and x is below it.

Variable Declarations

You can use the `let` keyword to declare variables. A variable is only visible inside the `{...}`-block it was defined in. When compiling to the EVM, a new stack slot is created that is reserved for the variable and automatically removed again when the end of the block is reached. You can provide an initial value for the variable. If you do not provide a value, the variable will be initialized to zero.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions `mstore`, `mload`, `sstore` and `sload`. Future dialects might introduce specific types for such pointers.

When a variable is referenced, its current value is copied. For the EVM, this translates to a `DUP` instruction.

```
{
  let zero := 0
  let v := calldataload(zero)
  {
    let y := add(sload(v), 1)
    v := y
  } // y is "deallocated" here
  sstore(v, zero)
} // v and zero are "deallocated" here
```

If the declared variable should have a type different from the default type, you denote that following a colon. You can also declare multiple variables in one statement when you assign from a function call that returns multiple values.

```
// This will not compile (u32 and u256 type not implemented yet)
{
  let zero:u32 := 0:u32
  let v:u256, t:u32 := f()
  let x, y := g()
}
```

Depending on the optimiser settings, the compiler can free the stack slots already after the variable has been used for the last time, even though it is still in scope.

Assignments

Variables can be assigned to after their definition using the `:=` operator. It is possible to assign multiple variables at the same time. For this, the number and types of the values have to match. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables. The same variable may not occur multiple times on the left-hand side of an assignment, e.g. `x, x := f()` is invalid.

```
let v := 0
// re-assign v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assign multiple values
v, t := f()
```

If

The if statement can be used for conditionally executing code. No “else” block can be defined. Consider using “switch” instead (see below) if you need multiple alternatives.

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

The curly braces for the body are required.

Switch

You can use a switch statement as an extended version of the if statement. It takes the value of an expression and compares it to several literal constants. The branch corresponding to the matching constant is taken. Contrary to other programming languages, for safety reasons, control flow does not continue from one case to the next. There can be a fallback or default case called `default` which is taken if none of the literal constants matches.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases is not enclosed by curly braces, but the body of a case does require them.

Loops

Yul supports for-loops which consist of a header containing an initializing part, a condition, a post-iteration part and a body. The condition has to be an expression, while the other three are blocks. If the initializing part declares any variables at the top level, the scope of these variables extends to all other parts of the loop.

The `break` and `continue` statements can be used in the body to exit the loop or skip to the post-part, respectively.

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be used as a replacement for while loops: Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Function Declarations

Yul allows the definition of functions. These should not be confused with functions in Solidity since they are never part of an external interface of a contract and are part of a namespace separate from the one for Solidity functions.

For the EVM, Yul functions take their arguments (and a return PC) from the stack and also put the results onto the stack. User-defined functions and built-in functions are called in exactly the same way.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function.

Functions declare parameters and return variables, similar to Solidity. To return a value, you assign it to the return variable(s).

If you call a function that returns multiple values, you have to assign them to multiple variables using `a, b := f(x)` or `let a, b := f(x)`.

The `leave` statement can be used to exit the current function. It works like the `return` statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

Note that the EVM dialect has a built-in function called `return` that quits the full execution context (internal message call) and not just the current yul function.

The following example implements the power function by square-and-multiply.

```
{
    function power(base, exponent) -> result {
```

(continues on next page)

(continued from previous page)

```

switch exponent
case 0 { result := 1 }
case 1 { result := base }
default {
    result := power(mul(base, base), div(exponent, 2))
    switch mod(exponent, 2)
        case 1 { result := mul(base, result) }
    }
}
}

```

3.32.5 Specification of Yul

This chapter describes Yul code formally. Yul code is usually placed inside Yul objects, which are explained in their own chapter.

```

Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =

```

(continues on next page)

(continued from previous page)

```

Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_] [a-zA-Z_$0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\r\n\\"] | '\\'.)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9] +

```

Restrictions on the Grammar

Apart from those directly imposed by the grammar, the following restrictions apply:

Switches must have at least one case (including the default case). All case values need to have the same type and distinct values. If all possible values of the expression type are covered, a default case is not allowed (i.e. a switch with a bool expression that has both a true and a false case do not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return variables of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed. The same variable name cannot occur more than once in the left-hand-side of an assignment or variable declaration.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

A `continue` or `break` statement can only be used inside the body of a for-loop, as follows. Consider the innermost loop that contains the statement. The loop and the statement must be in the same function, or both must be at the top level. The statement must be in the loop's body block; it cannot be in the loop's initialization block or update block. It is worth emphasizing that this restriction applies just to the innermost loop that contains the `continue` or `break` statement: this innermost loop, and therefore the `continue` or `break` statement, may appear anywhere in an outer loop, possibly in an outer loop's initialization block or update block. For example, the following is legal, because the `break` occurs in the body block of the inner loop, despite also occurring in the update block of the outer loop:

```

for {} true { for {} true {} { break } }
{
}

```

The condition part of the for-loop has to evaluate to exactly one value.

The `leave` statement can only be used inside a function.

Functions cannot be defined anywhere inside for loop init blocks.

Literals cannot be larger than their type. The largest type defined is 256-bit wide.

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations ([FunctionDefinition](#), [VariableDeclaration](#)) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks): Functions are visible in the whole block (even before their definitions) while variables are only visible starting from the statement after the [VariableDeclaration](#).

In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

As an exception to the general scoping rule, the scope of the “init” part of the for-loop (the first block) extends across all other parts of the for loop. This means that variables (and functions) declared in the init part (but not inside a block inside the init part) are visible in all other parts of the for-loop.

Identifiers declared in the other parts of the for loop respect the regular syntactical scoping rules.

This means a for-loop of the form `for { I... } C { P... } { B... }` is equivalent to `{ I... for {} C { P... } { B... } }`.

The parameters and return parameters of functions are visible in the function body and their names have to be distinct.

Inside functions, it is not possible to reference a variable that was declared outside of that function.

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not possible to reference it because it was declared outside the current function.

Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. As built-in functions can have side effects, E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM).

If the AST node is a statement, E returns the two state objects and a “mode”, which is used for the `break`, `continue` and `leave` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state L is a mapping of identifiers i to values v, denoted as $L[i] = v$.

For an identifier v, let \$v be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```

E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise

```

(continues on next page)

(continued from previous page)

```

        G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L, mode = E(G, L, i1, ..., in)
        // mode has to be regular or leave due to the syntactic restrictions
        if mode is leave then
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G2, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =

```

(continues on next page)

(continued from previous page)

```

let G0, L0, v = E(G, L, condition)
// i = 1 .. n
// Evaluate literals, context doesn't matter
let _, _, v1 = E(G0, L0, l1)
...
let _, _, vn = E(G0, L0, ln)
if there exists smallest i such that vi = v:
    E(G0, L0, sti)
else:
    E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
...
G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
be the function of name $fname visible at the point of the call.
Let L' be a new local state such that
L'[$parami] = vi and L'[$reti] = 0 for all i.
Let G'', L'', mode = E(Gn, L', block)
G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    where str is the string evaluation function,
    which for the EVM dialect is defined in the section 'Literals' above
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal evaluation function,
    which turns a sequence of hexadecimal digits into their big endian value
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value

```

EVM Dialect

The default dialect of Yul currently is the EVM dialect for the currently selected version of the EVM, with a version of the EVM. The only type available in this dialect is u256, the 256-bit native type of the Ethereum Virtual Machine. Since it is the default type of this dialect, it can be omitted.

The following table lists all builtin functions (depending on the EVM version) and provides a short description of the semantics of the function / opcode. This document does not want to be a full description of the Ethereum virtual machine. Please refer to a different document if you are interested in the precise semantics.

Opcodes marked with - do not return a result and all others return exactly one value. Opcodes marked with F, H, B, C, I and L are present since Frontier, Homestead, Byzantium, Constantinople, Istanbul or London respectively.

In the following, `mem[a...b]` signifies the bytes of memory starting at position a up to but not including position b and `storage[p]` signifies the storage contents at slot p.

Since Yul manages local variables and control-flow, opcodes that interfere with these features are not available. This includes the `dup` and `swap` instructions as well as `jump` instructions, labels and the `push` instructions.

Instruction			Explanation
stop()	-	F	stop execution, identical to return(0, 0)
add(x, y)		F	$x + y$
sub(x, y)		F	$x - y$
mul(x, y)		F	$x * y$
div(x, y)		F	x / y or 0 if $y == 0$
sdiv(x, y)		F	x / y , for signed numbers in two's complement, 0 if $y == 0$
mod(x, y)		F	$x \% y$, 0 if $y == 0$
smod(x, y)		F	$x \% y$, for signed numbers in two's complement, 0 if $y == 0$
exp(x, y)		F	x to the power of y
not(x)		F	bitwise “not” of x (every bit of x is negated)
lt(x, y)		F	1 if $x < y$, 0 otherwise
gt(x, y)		F	1 if $x > y$, 0 otherwise
slt(x, y)		F	1 if $x < y$, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		F	1 if $x > y$, 0 otherwise, for signed numbers in two's complement
eq(x, y)		F	1 if $x == y$, 0 otherwise
iszero(x)		F	1 if $x == 0$, 0 otherwise
and(x, y)		F	bitwise “and” of x and y
or(x, y)		F	bitwise “or” of x and y
xor(x, y)		F	bitwise “xor” of x and y
byte(n, x)		F	nth byte of x , where the most significant byte is the 0th byte
shl(x, y)		C	logical shift left y by x bits
shr(x, y)		C	logical shift right y by x bits
sar(x, y)		C	signed arithmetic shift right y by x bits
addmod(x, y, m)		F	$(x + y) \% m$ with arbitrary precision arithmetic, 0 if $m == 0$
mulmod(x, y, m)		F	$(x * y) \% m$ with arbitrary precision arithmetic, 0 if $m == 0$
signextend(i, x)		F	sign extend from $(i*8+7)$ th bit counting from least significant
keccak256(p, n)		F	keccak(mem[p...(p+n)])
pc()		F	current position in code
pop(x)	-	F	discard value x
mload(p)		F	mem[p...(p+32)]
mstore(p, v)	-	F	mem[p...(p+32)] := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (only modifies a single byte)
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	size of memory, i.e. largest accessed memory index
gas()		F	gas still available to execution
address()		F	address of the current contract / execution context
balance(a)		F	wei balance at address a
selfbalance()		I	equivalent to balance(address()), but cheaper
caller()		F	call sender (excluding delegatecall)
callvalue()		F	wei sent together with the current call
calldataload(p)		F	call data starting from position p (32 bytes)
calldatasize()		F	size of call data in bytes
calldatacopy(t, f, s)	-	F	copy s bytes from calldata at position f to mem at position t
codesize()		F	size of the code of the current contract / execution context
codecopy(t, f, s)	-	F	copy s bytes from code at position f to mem at position t
extcodesize(a)		F	size of the code at address a
extcodecopy(a, t, f, s)	-	F	like codecopy(t, f, s) but take code at address a
returndatasize()		B	size of the last returndata
returndatacopy(t, f, s)	-	B	copy s bytes from returndata at position f to mem at position t

Instruction		Explanation
extcodehash(a)	C	code hash of address a
create(v, p, n)	F	create new contract with code mem[p...(p+n)) and send v wei and return the new address
create2(v, p, n, s)	C	create new contract with code mem[p...(p+n)) at address keccak256(0xff . this . s)
call(g, a, v, in, insize, out, outsize)	F	call contract at address a with input mem[in...(in+insize)) providing g gas and v value
callcode(g, a, v, in, insize, out, outsize)	F	identical to call but only use the code from a and stay in the context of the current contract
delegatecall(g, a, in, insize, out, outsize)	H	identical to callcode but also keep caller and callvalue See more
staticcall(g, a, in, insize, out, outsize)	B	identical to call(g, a, 0, in, insize, out, outsize) but do not allow selfdestruct
return(p, s)	- F	end execution, return data mem[p...(p+s))
revert(p, s)	- B	end execution, revert state changes, return data mem[p...(p+s))
selfdestruct(a)	- F	end execution, destroy current contract and send funds to a
invalid()	- F	end execution with invalid instruction
log0(p, s)	- F	log without topics and data mem[p...(p+s))
log1(p, s, t1)	- F	log with topic t1 and data mem[p...(p+s))
log2(p, s, t1, t2)	- F	log with topics t1, t2 and data mem[p...(p+s))
log3(p, s, t1, t2, t3)	- F	log with topics t1, t2, t3 and data mem[p...(p+s))
log4(p, s, t1, t2, t3, t4)	- F	log with topics t1, t2, t3, t4 and data mem[p...(p+s))
chainid()	I	ID of the executing chain (EIP-1344)
basefee()	L	current block's base fee (EIP-3198 and EIP-1559)
origin()	F	transaction sender
gasprice()	F	gas price of the transaction
blockhash(b)	F	hash of block nr b - only for last 256 blocks excluding current
coinbase()	F	current mining beneficiary
timestamp()	F	timestamp of the current block in seconds since the epoch
number()	F	current block number
difficulty()	F	difficulty of the current block
gaslimit()	F	block gas limit of the current block

Note: The `call*` instructions use the `out` and `outsize` parameters to define an area in memory where the return or failure data is placed. This area is written to depending on how many bytes the called contract returns. If it returns more data, only the first `outsize` bytes are written. You can access the rest of the data using the `returndatcopy` opcode. If it returns less data, then the remaining bytes are not touched at all. You need to use the `returndatasize` opcode to check which part of this memory area contains the return data. The remaining bytes will retain their values as of before the call.

In some internal dialects, there are additional functions:

datasize, dataoffset, datacopy

The functions `datasize(x)`, `dataoffset(x)` and `datacopy(t, f, l)` are used to access other parts of a Yul object. `datasize` and `dataoffset` can only take string literals (the names of other objects) as arguments and return the size and offset in the data area, respectively. For the EVM, the `datacopy` function is equivalent to `codecopy`.

setimmutable, loadimmutable

The functions `setimmutable(offset, "name", value)` and `loadimmutable("name")` are used for the immutable mechanism in Solidity and do not nicely map to pure Yul. The call to `setimmutable(offset, "name", value)` assumes that the runtime code of the contract containing the given named immutable was copied to memory at offset `offset` and will write `value` to all positions in memory (relative to `offset`) that contain the placeholder that was generated for calls to `loadimmutable("name")` in the runtime code.

linkersymbol

The function `linkersymbol("library_id")` is a placeholder for an address literal to be substituted by the linker. Its first and only argument must be a string literal and uniquely represents the address to be inserted. Identifiers can be arbitrary but when the compiler produces Yul code from Solidity sources, it uses a library name qualified with the name of the source unit that defines that library. To link the code with a particular library address, the same identifier must be provided to the `--libraries` option on the command line.

For example this code

```
let a := linkersymbol("file.sol:Math")
```

is equivalent to

```
let a := 0x1234567890123456789012345678901234567890
```

when the linker is invoked with `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890"` option.

See [Using the Commandline Compiler](#) for details about the Solidity linker.

memoryguard

This function is available in the EVM dialect with objects. The caller of `let ptr := memoryguard(size)` (where `size` has to be a literal number) promises that they only use memory in either the range `[0, size]` or the unbounded range starting at `ptr`.

Since the presence of a `memoryguard` call indicates that all memory access adheres to this restriction, it allows the optimizer to perform additional optimization steps, for example the stack limit evader, which attempts to move stack variables that would otherwise be unreachable to memory.

The Yul optimizer promises to only use the memory range `[size, ptr)` for its purposes. If the optimizer does not need to reserve any memory, it holds that `ptr == size`.

`memoryguard` can be called multiple times, but needs to have the same literal as argument within one Yul subobject. If at least one `memoryguard` call is found in a subobject, the additional optimiser steps will be run on it.

verbatim

The set of `verbatim...` builtin functions lets you create bytecode for opcodes that are not known to the Yul compiler. It also allows you to create bytecode sequences that will not be modified by the optimizer.

The functions are `verbatim_<n>i_<m>o("<data>", ...)`, where

- `n` is a decimal between 0 and 99 that specifies the number of input stack slots / variables
- `m` is a decimal between 0 and 99 that specifies the number of output stack slots / variables
- `data` is a string literal that contains the sequence of bytes

If you for example want to define a function that multiplies the input by two, without the optimizer touching the constant two, you can use

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

This code will result in a `dup1` opcode to retrieve `x` (the optimizer might directly re-use result of the `calldataload` opcode, though) directly followed by `600202`. The code is assumed to consume the copied value of `x` and produce the result on the top of the stack. The compiler then generates code to allocate a stack slot for `double` and store the result there.

As with all opcodes, the arguments are arranged on the stack with the leftmost argument on the top, while the return values are assumed to be laid out such that the rightmost variable is at the top of the stack.

Since `verbatim` can be used to generate arbitrary opcodes or even opcodes unknown to the Solidity compiler, care has to be taken when using `verbatim` together with the optimizer. Even when the optimizer is switched off, the code generator has to determine the stack layout, which means that e.g. using `verbatim` to modify the stack height can lead to undefined behaviour.

The following is a non-exhaustive list of restrictions on `verbatim` bytecode that are not checked by the compiler. Violations of these restrictions can result in undefined behaviour.

- Control-flow should not jump into or out of `verbatim` blocks, but it can jump within the same `verbatim` block.
- Stack contents apart from the input and output parameters should not be accessed.
- The stack height difference should be exactly $m - n$ (output slots minus input slots).
- `Verbatim` bytecode cannot make any assumptions about the surrounding bytecode. All required parameters have to be passed in as stack variables.

The optimizer does not analyze `verbatim` bytecode and always assumes that it modifies all aspects of state and thus can only do very few optimizations across `verbatim` function calls.

The optimizer treats `verbatim` bytecode as an opaque block of code. It will not split it but might move, duplicate or combine it with identical `verbatim` bytecode blocks. If a `verbatim` bytecode block is unreachable by the control-flow, it can be removed.

Warning: During discussions about whether or not EVM improvements might break existing smart contracts, features inside `verbatim` cannot receive the same consideration as those used by the Solidity compiler itself.

Note: To avoid confusion, all identifiers starting with the string `verbatim` are reserved and cannot be used for user-defined identifiers.

3.32.6 Specification of Yul Object

Yul objects are used to group named code and data sections. The functions `datasize`, `dataoffset` and `datacopy` can be used to access these sections from within code. Hex strings can be used to specify data in hex encoding, regular strings in native encoding. For code, `datacopy` will access its assembled binary representation.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\' ([0-9a-fA-F]{2})* '\\')
StringLiteral = '"' ([^"\r\n\\"] | '\\'.)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

Note: An object with a name that ends in `_deployed` is treated as deployed code by the Yul optimizer. The only consequence of this is a different gas cost heuristic in the optimizer.

Note: Data objects or sub-objects whose names contain a `.` can be defined but it is not possible to access them through `datasize`, `dataoffset` or `datacopy` because `.` is used as a separator to access objects inside another object.

Note: The data object called `".metadata"` has a special meaning: It cannot be accessed from code and is always appended to the very end of the bytecode, regardless of its position in the object.

Other data objects with special significance might be added in the future, but their names will always start with a `..`

An example Yul Object is shown below:

```
// A contract consists of a single object with sub-objects representing
// the code to be deployed or other contracts it can create.
// The single "code" node is the executable code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset / ...
// datasize
// The current object, sub-objects and data items inside the current object
// are in scope.
object "Contract1" {
    // This is the constructor code of the contract.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // first create "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // This will turn into codecopy for EVM
        datacopy(offset, dataoffset("Contract2"), size)
        // constructor parameter is a single number 0x1234
    }
}
```

(continues on next page)

(continued from previous page)

```

mstore(add(offset, size), 0x1234)
pop(create(offset, add(size, 32), 0))

// now return the runtime object (the currently
// executing code is the constructor code)
size := datasize("Contract1_deployed")
offset := allocate(size)
// This will turn into a memory->memory copy for Ewasm and
// a codecopy for EVM
datacopy(offset, dataoffset("Contract1_deployed"), size)
return(offset, size)
}

data "Table2" hex"4123"

object "Contract1_deployed" {
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // runtime code

        mstore(0, "Hello, World!")
        return(0, 0x20)
    }
}

// Embedded object. Use case is that the outside is a factory contract,
// and Contract2 is the code to be created by the factory
object "Contract2" {
    code {
        // code here ...
    }

    object "Contract2_deployed" {
        code {
            // code here ...
        }
    }

    data "Table1" hex"4123"
}
}

```

3.32.7 Yul Optimizer

The Yul optimizer operates on Yul code and uses the same language for input, output and intermediate states. This allows for easy debugging and verification of the optimizer.

Please refer to the general [optimizer documentation](#) for more details about the different optimization stages and how to use the optimizer.

If you want to use Solidity in stand-alone Yul mode, you activate the optimizer using `--optimize` and optionally specify the [expected number of contract executions](#) with `--optimize-runs`:

```
solc --strict-assembly --optimize --optimize-runs 200
```

In Solidity mode, the Yul optimizer is activated together with the regular optimizer.

Optimization Step Sequence

By default the Yul optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new optimization opportunities for others that were already applied so repeating steps is often beneficial. By enclosing part of the sequence in square brackets ([]) you tell the optimizer to repeatedly apply that part until it no longer improves the size of the resulting assembly. You can use brackets multiple times in a single sequence but they cannot be nested.

The following optimization steps are available:

Abbreviation	Full name
f	BlockFlattener
l	CircularReferencesPruner
c	CommonSubexpressionEliminator
C	ConditionalSimplifier
U	ConditionalUnsimplifier
n	ControlFlowSimplifier
D	DeadCodeEliminator
v	EquivalentFunctionCombiner
e	ExpressionInliner
j	ExpressionJoiner
s	ExpressionSimplifier
x	ExpressionSplitter
I	ForLoopConditionIntoBody
O	ForLoopConditionOutOfBody
o	ForLoopInitRewriter
i	FullInliner
g	FunctionGrouper
h	FunctionHoister
F	FunctionSpecializer
T	LiteralRematerialiser
L	LoadResolver
M	LoopInvariantCodeMotion
r	RedundantAssignEliminator

continues on next page

Table 2 – continued from previous page

Abbreviation	Full name
R	ReasoningBasedSimplifier - highly experimental
m	Rematerialiser
V	SSAREverser
a	SSATransform
t	StructuralSimplifier
u	UnusedPruner
p	UnusedFunctionParameterPruner
d	VarDeclInitializer

Some steps depend on properties ensured by `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter`. For this reason the Yul optimizer always applies them before applying any steps supplied by the user.

The ReasoningBasedSimplifier is an optimizer step that is currently not enabled in the default set of steps. It uses an SMT solver to simplify arithmetic expressions and boolean conditions. It has not received thorough testing or validation yet and can produce non-reproducible results, so please use with care!

3.32.8 Complete ERC20 Example

```

object "Token" {
    code {
        // Store the creator in slot zero.
        sstore(0, caller())

        // Deploy the contract
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Protection against sending Ether
            require(iszero(callvalue()))

            // Dispatcher
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ {
                returnUint(balanceOf(decodeAsAddress(0)))
            }
            case 0x18160ddd /* "totalSupply()" */ {
                returnUint(totalSupply())
            }
            case 0xa9059ccb /* "transfer(address,uint256)" */ {
                transfer(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
                transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
                returnTrue()
            }
            case 0x095ea7b3 /* "approve(address,uint256)" */ {
                approve(decodeAsAddress(0), decodeAsUint(1))
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        returnTrue()
    }
case 0xdd62ed3e /* "allowance(address,address)" */ {
    returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
}
case 0x40c10f19 /* "mint(address,uint256)" */ {
    mint(decodeAsAddress(0), decodeAsUint(1))
    returnTrue()
}
default {
    revert(0, 0)
}

function mint(account, amount) {
    require(calledByOwner())

    mintTokens(amount)
    addToBalance(account, amount)
    emitTransfer(0, account, amount)
}

function transfer(to, amount) {
    executeTransfer(caller(), to, amount)
}

function approve(spender, amount) {
    revertIfZeroAddress(spender)
    setAllowance(caller(), spender, amount)
    emitApproval(caller(), spender, amount)
}

function transferFrom(from, to, amount) {
    decreaseAllowanceBy(from, caller(), amount)
    executeTransfer(from, to, amount)
}

function executeTransfer(from, to, amount) {
    revertIfZeroAddress(to)
    deductFromBalance(from, amount)
    addToBalance(to, amount)
    emitTransfer(from, to, amount)
}

/* ----- calldata decoding functions ----- */
function selector() -> s {
    s := div(calldataload(0),
→0x1000000000000000000000000000000000000000000000000000000000000000)
}

function decodeAsAddress(offset) -> v {
    v := decodeAsUint(offset)
    if iszero(iszero(and(v,
→not(0xffffffffffffffffffffffffff)))) {
        revert(0, 0)
    }
}

```

(continues on next page)

(continued from previous page)

```

        }
    }

function decodeAsUint(offset) -> v {
    let pos := add(4, mul(offset, 0x20))
    if lt(calldatasize(), add(pos, 0x20)) {
        revert(0, 0)
    }
    v := calldataload(pos)
}

/* ----- calldata encoding functions ----- */
function returnUint(v) {
    mstore(0, v)
    return(0, 0x20)
}
function returnTrue() {
    returnUint(1)
}

/* ----- events ----- */
function emitTransfer(from, to, amount) {
    let signatureHash := 0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
    emitEvent(signatureHash, from, to, amount)
}
function emitApproval(from, spender, amount) {
    let signatureHash := 0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
    emitEvent(signatureHash, from, spender, amount)
}
function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
    mstore(0, nonIndexed)
    log3(0, 0x20, signatureHash, indexed1, indexed2)
}

/* ----- storage layout ----- */
function ownerPos() -> p { p := 0 }
function totalSupplyPos() -> p { p := 1 }
function accountToStorageOffset(account) -> offset {
    offset := add(0x1000, account)
}
function allowanceStorageOffset(account, spender) -> offset {
    offset := accountToStorageOffset(account)
    mstore(0, offset)
    mstore(0x20, spender)
    offset := keccak256(0, 0x40)
}

/* ----- storage access ----- */
function owner() -> o {
    o := sload(ownerPos())
}
function totalSupply() -> supply {

```

(continues on next page)

(continued from previous page)

```

        supply := sload(totalSupplyPos())
    }
    function mintTokens(amount) {
        sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
    }
    function balanceOf(account) -> bal {
        bal := sload(accountToStorageOffset(account))
    }
    function addToBalance(account, amount) {
        let offset := accountToStorageOffset(account)
        sstore(offset, safeAdd(sload(offset), amount))
    }
    function deductFromBalance(account, amount) {
        let offset := accountToStorageOffset(account)
        let bal := sload(offset)
        require(lte(amount, bal))
        sstore(offset, sub(bal, amount))
    }
    function allowance(account, spender) -> amount {
        amount := sload(allowanceStorageOffset(account, spender))
    }
    function setAllowance(account, spender, amount) {
        sstore(allowanceStorageOffset(account, spender), amount)
    }
    function decreaseAllowanceBy(account, spender, amount) {
        let offset := allowanceStorageOffset(account, spender)
        let currentAllowance := sload(offset)
        require(lte(amount, currentAllowance))
        sstore(offset, sub(currentAllowance, amount))
    }

    /* ----- utility functions ----- */
    function lte(a, b) -> r {
        r := iszero(gt(a, b))
    }
    function gte(a, b) -> r {
        r := iszero lt(a, b)
    }
    function safeAdd(a, b) -> r {
        r := add(a, b)
        if or(lt(r, a), lt(r, b)) { revert(0, 0) }
    }
    function calledByOwner() -> cbo {
        cbo := eq(owner(), caller())
    }
    function revertIfZeroAddress(addr) {
        require(addr)
    }
    function require(condition) {
        if iszero(condition) { revert(0, 0) }
    }
}

```

(continues on next page)

(continued from previous page)

```
}
```

3.33 Style Guide

3.33.1 Introduction

This guide is intended to provide coding conventions for writing Solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from python's [pep8](#) style guide.

The goal of this guide is *not* to be the right way or the best way to write Solidity code. The goal of this guide is *consistency*. A quote from python's [pep8](#) captures this concept well.

Note: A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: **know when to be inconsistent** – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

3.33.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

Blank Lines

Surround top level declarations in Solidity source with two blank lines.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
```

(continues on next page)

(continued from previous page)

```
contract B {
    // ...
}

contract C {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}

contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}

contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

Maximum Line Length

Keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `) ;`, should be placed on the final line by itself.

Function Calls

Yes:

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No:

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
                            longArgument2,
                            longArgument3
);
```

(continues on next page)

(continued from previous page)

```
thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(argument1,
                                                               argument2,
                                                               argument3,
                                                               argument4);
```

Event Definitions and Event Emitters

Yes:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}

import "./Owned.sol";

contract B is Owned {
```

(continues on next page)

(continued from previous page)

```
// ...
}
```

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...
}
```

(continues on next page)

(continued from previous page)

```
// Private functions
// ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // External functions
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() {
        // ...
    }

    // Internal functions
    // ...
}
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() public { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin) public;
```

No:

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another:

Yes:

```
x = 1;
y = 2;
longVariable = 3;
```

No:

```
x           = 1;
y           = 2;
longVariable = 3;
```

Don't include a whitespace in the receive and fallback functions:

Yes:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

No:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

Control Structures

The braces denoting the body of a contract, library, functions and structs should:

- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be preceded by a single space.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...) {
    ...
}

while(...){

for (...) {
    ...;
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
}
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}

if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceded by a single space.

Yes:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}
```

(continues on next page)

(continued from previous page)

```
function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yes:

```
function balance(uint from) public view override returns (uint) {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

No:

```
function balance(uint from) public override view returns (uint) {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

For long function declarations, it is recommended to drop each argument onto its own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the

same indentation level as the function declaration.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its own line.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}
```

(continues on next page)

(continued from previous page)

```
function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z
)
public
onlyOwner
priced
returns (address)
{
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
public
onlyOwner
priced
returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
public onlyOwner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
public
onlyOwner
priced
returns (address) {
    doSomething();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the [Maximum Line Length](#) section.

Yes:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
public
returns (
    address someAddressName,
    uint256 LongArgument,
    uint256 Argument
```

(continues on next page)

(continued from previous page)

```

    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}

```

No:

```

function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
public
returns (address someAddressName,
           uint256 LongArgument,
           uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
             veryLongReturnArg1,
             veryLongReturnArg1);
}

```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}

contract C {
    constructor(uint, uint) {
    }
}

contract D {
    constructor(uint) {
    }
}

```

(continues on next page)

(continued from previous page)

}

```
contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
        x = param5;
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}

contract C {
    constructor(uint, uint) {
    }
}

contract D {
    constructor(uint) {
    }
}

contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

(continues on next page)

(continued from previous page)

```
contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() public { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgment as this guide does not try to cover all possible permutations for function declarations.

Mappings

In variable declarations, do not separate the keyword mapping from its type by a space. Do not separate any nested mapping keyword from its type by whitespace.

Yes:

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

No:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

Other Recommendations

- Strings should be quoted with double-quotes instead of single-quotes.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Surround operators with a single space on either side.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator:

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

3.33.3 Order of Layout

Layout contract elements in the following order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Note: It might be clearer to declare types close to their use in events or state variables.

3.33.4 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supersede any conventions outlined in this document.

Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)

Note: When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus `HTTPServerError` is better than `HttpServerError`. When using initialisms in mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus `xmlHTTPRequest` is better than `XMLHTTPRequest`.

Names to Avoid

- l - Lowercase letter el
- O - Uppercase letter oh
- I - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

Contract and Library Names

- Contracts and libraries should be named using the CapWords style. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
- Contract and library names should also match their filenames.
- If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is Congress and the library name is Owned, then their associated filenames should be `Congress.sol` and `Owned.sol`.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        -
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract Congress is Owned, TokenRecipient {
```

(continues on next page)

(continued from previous page)

```
//...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        -
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in Congress.sol:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;

import "./owned.sol";

contract Congress is owned, tokenRecipient {
    //...
}
```

Struct Names

Structs should be named using the CapWords style. Examples: `MyCoin`, `Position`, `PositionXY`.

Event Names

Events should be named using the CapWords style. Examples: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Function Names

Functions should use mixedCase. Examples: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Function Argument Names

Function arguments should use mixedCase. Examples: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

Local and State Variable Names

Use mixedCase. Examples: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constants

Constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Names

Use mixedCase. Examples: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Avoiding Naming Collisions

- `singleTrailingUnderscore_`

This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

3.33.5 NatSpec

Solidity contracts can also contain NatSpec comments. They are written with a triple slash (///) or a double asterisk block (/** ... */) and they should be used directly above function declarations or statements.

For example, the contract from *a simple smart contract* with the comments added looks like the one below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;

    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
    function get() public view returns (uint) {
        return storedData;
    }
}
```

It is recommended that Solidity contracts are fully annotated using *NatSpec* for all public interfaces (everything in the ABI).

Please see the section about *NatSpec* for a detailed explanation.

3.34 Common Patterns

3.34.1 Withdrawal from Contracts

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct `transfer` call, this is not recommended as it introduces a potential security risk. You may read more about this on the [Security Considerations](#) page.

The following is an example of the withdrawal pattern in practice in a contract where the goal is to send the most money to the contract in order to become the “richest”, inspired by [King of the Ether](#).

In the following contract, if you are no longer the richest, you receive the funds of the person who is now the richest.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    /// The amount of Ether sent was not higher than
    /// the currently highest amount.
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

This is as opposed to the more intuitive sending pattern:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// The amount of Ether sent was not higher than
    /// the currently highest amount.
    error NotEnoughEther();

    constructor() payable {
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
```

(continues on next page)

(continued from previous page)

```

if (msg.value <= mostSent) revert NotEnoughEther();
// This line can cause problems (explained below).
richest.transfer(msg.value);
richest = payable(msg.sender);
mostSent = msg.value;
}
}

```

Notice that, in this example, an attacker could trap the contract into an unusable state by causing `richest` to be the address of a contract that has a receive or fallback function which fails (e.g. by using `revert()` or by just consuming more than the 2300 gas stipend transferred to them). That way, whenever `transfer` is called to deliver funds to the “poisoned” contract, it will fail and thus also `becomeRichest` will fail, with the contract being stuck forever.

In contrast, if you use the “withdraw” pattern from the first example, the attacker can only cause his or her own withdraw to fail and not the rest of the contract’s workings.

3.34.2 Restricting Access

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract’s state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract’s state by **other contracts**. That is actually the default unless you declare your state variables `public`.

Furthermore, you can restrict who can make modifications to your contract’s state or call your contract’s functions and this is what this section is about.

The use of **function modifiers** makes these restrictions highly readable.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // creating this contract.
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;

    // Now follows a list of errors that
    // this contract can generate together
    // with a textual explanation in special
    // comments.

    /// Sender not authorized for this
    /// operation.
    error Unauthorized();

    /// Function called too early.
    error TooEarly();

    /// Not enough Ether sent with function call.

```

(continues on next page)

(continued from previous page)

```

error NotEnoughEther();

// Modifiers can be used to change
// the body of a function.
// If this modifier is used, it will
// prepend a check that only passes
// if the function is called from
// a certain address.
modifier onlyBy(address account)
{
    if (msg.sender != account)
        revert Unauthorized();
    // Do not forget the "_"! It will
    // be replaced by the actual function
    // body when the modifier is used.
    -;
}

/// Make `newOwner` the new owner of this
/// contract.
function changeOwner(address newOwner)
public
onlyBy(owner)
{
    owner = newOwner;
}

modifier onlyAfter(uint time) {
    if (block.timestamp < time)
        revert TooEarly();
    -;
}

/// Erase ownership information.
/// May only be called 6 weeks after
/// the contract has been created.
function disown()
public
onlyBy(owner)
onlyAfter(creationTime + 6 weeks)
{
    delete owner;
}

// This modifier requires a certain
// fee being associated with a function call.
// If the caller sent too much, he or she is
// refunded, but only after the function body.
// This was dangerous before Solidity version 0.4.0,
// where it was possible to skip the part after `_;`.
modifier costs(uint amount) {
    if (msg.value < amount)

```

(continues on next page)

(continued from previous page)

```

        revert NotEnoughEther();

    -;
    if (msg.value > amount)
        payable(msg.sender).transfer(msg.value - amount);
}

function forceOwnerChange(address newOwner)
public
payable
costs(200 ether)
{
    owner = newOwner;
    // just some example condition
    if (uint160(owner) & 0 == 1)
        // This did not refund for Solidity
        // before version 0.4.0.
        return;
    // refund overpaid fees
}
}

```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

3.34.3 State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

An example for this is a blind auction contract which starts in the stage “accepting blinded bids”, then transitions to “revealing bids” which is ended by “determine auction outcome”.

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier `atStage` ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier `timedTransitions`, which should be used for all functions.

Note: Modifier Order Matters. If `atStage` is combined with `timedTransitions`, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier `transitionNext` can be used to automatically go to the next stage when the function finishes.

Note: Modifier May be Skipped. This only applies to Solidity before version 0.4.0: Since modifiers are applied by simply replacing code and not by using a function call, the code in the `transitionNext` modifier can be skipped if the

function itself uses `return`. If you want to do that, make sure to call `nextStage` manually from those functions. Starting with version 0.4.0, modifier code will run even if the function explicitly returns.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    /// Function cannot be called at this time.
    error FunctionInvalidAtThisStage();

    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages stage_) {
        if (stage != stage_)
            revert FunctionInvalidAtThisStage();
        ;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
            block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
            block.timestamp >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
        ;
    }

    // Order of the modifiers matters here!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
```

(continues on next page)

(continued from previous page)

```
{  
    // We will not implement that here  
}  
  
function reveal()  
    public  
    timedTransitions  
    atStage(Stages.RevealBids)  
{  
}  
  
// This modifier goes to the next stage  
// after the function is done.  
modifier transitionNext()  
{  
    -;  
    nextStage();  
}  
  
function g()  
    public  
    timedTransitions  
    atStage(Stages.AnotherStage)  
    transitionNext  
{  
}  
  
function h()  
    public  
    timedTransitions  
    atStage(Stages.AreWeDoneYet)  
    transitionNext  
{  
}  
  
function i()  
    public  
    timedTransitions  
    atStage(Stages.Finished)  
{  
}  
}
```

3.35 List of Known Bugs

Below, you can find a JSON-formatted list of some of the known security-relevant bugs in the Solidity compiler. The file itself is hosted in the [Github repository](#). The list stretches back as far as version 0.3.0, bugs known to be present only in versions preceding that are not listed.

There is another file called `bugs_by_version.json`, which can be used to check which bugs affect a specific version of the compiler.

Contract source verification tools and also other tools interacting with contracts should consult this list according to the following criteria:

- It is mildly suspicious if a contract was compiled with a nightly compiler version instead of a released version. This list does not keep track of unreleased or nightly versions.
- It is also mildly suspicious if a contract was compiled with a version that was not the most recent at the time the contract was created. For contracts created from other contracts, you have to follow the creation chain back to a transaction and use the date of that transaction as creation date.
- It is highly suspicious if a contract was compiled with a compiler that contains a known bug and the contract was created at a time where a newer compiler version containing a fix was already released.

The JSON file of known bugs below is an array of objects, one for each bug, with the following keys:

uid

Unique identifier given to the bug in the form of `SOL-<year>-<number>`. It is possible that multiple entries exists with the same uid. This means multiple version ranges are affected by the same bug.

name

Unique name given to the bug

summary

Short description of the bug

description

Detailed description of the bug

link

URL of a website with more detailed information, optional

introduced

The first published compiler version that contained the bug, optional

fixed

The first published compiler version that did not contain the bug anymore

publish

The date at which the bug became known publicly, optional

severity

Severity of the bug: very low, low, medium, high. Takes into account discoverability in contract tests, likelihood of occurrence and potential damage by exploits.

conditions

Conditions that have to be met to trigger the bug. The following keys can be used: `optimizer`, Boolean value which means that the optimizer has to be switched on to enable the bug. `evmVersion`, a string that indicates which EVM version compiler settings trigger the bug. The string can contain comparison operators. For example, "`>=constantinople`" means that the bug is present when the EVM version is set to `constantinople` or later. If no conditions are given, assume that the bug is present.

check

This field contains different checks that report whether the smart contract contains the bug or not. The first type

of check are Javascript regular expressions that are to be matched against the source code (“source-regex”) if the bug is present. If there is no match, then the bug is very likely not present. If there is a match, the bug might be present. For improved accuracy, the checks should be applied to the source code after stripping comments. The second type of check are patterns to be checked on the compact AST of the Solidity program (“ast-compact-json-path”). The specified search query is a JsonPath expression. If at least one path of the Solidity AST matches the query, the bug is likely present.

```
[  
  {  
    "uid": "SOL-2022-5",  
    "name": "DirtyBytesArrayToStorage",  
    "summary": "Copying ``bytes`` arrays from memory or calldata to storage may  
    ↵ result in dirty storage values.",  
    "description": "Copying ``bytes`` arrays from memory or calldata to storage is  
    ↵ done in chunks of 32 bytes even if the length is not a multiple of 32. Thereby, extra  
    ↵ bytes past the end of the array may be copied from calldata or memory to storage.  
    ↵ These dirty bytes may then become observable after a ``.push()`` without arguments to  
    ↵ the bytes array in storage, i.e. such a push will not result in a zero value at the  
    ↵ end of the array as expected. This bug only affects the legacy code generation  
    ↵ pipeline, the new code generation pipeline via IR is not affected.",  
    "link": "https://blog.soliditylang.org/2022/06/15/dirty-bytes-array-to-storage-  
    ↵ bug/",  
    "introduced": "0.0.1",  
    "fixed": "0.8.15",  
    "severity": "low"  
  },  
  {  
    "uid": "SOL-2022-4",  
    "name": "InlineAssemblyMemorySideEffects",  
    "summary": "The Yul optimizer may incorrectly remove memory writes from inline  
    ↵ assembly blocks, that do not access solidity variables.",  
    "description": "The Yul optimizer considers all memory writes in the outermost  
    ↵ Yul block that are never read from as unused and removes them. This is valid when that  
    ↵ Yul block is the entire Yul program, which is always the case for the Yul code  
    ↵ generated by the new via-IR pipeline. Inline assembly blocks are never optimized in  
    ↵ isolation when using that pipeline. Instead they are optimized as a part of the whole  
    ↵ Yul input. However, the legacy code generation pipeline (which is still the default)  
    ↵ runs the Yul optimizer individually on an inline assembly block if the block does not  
    ↵ refer to any local variables defined in the surrounding Solidity code. Consequently,  
    ↵ memory writes in such inline assembly blocks are removed as well, if the written  
    ↵ memory is never read from in the same assembly block, even if the written memory is  
    ↵ accessed later, for example by a subsequent inline assembly block.",  
    "link": "https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-  
    ↵ effects-bug/",  
    "introduced": "0.8.13",  
    "fixed": "0.8.15",  
    "severity": "medium",  
    "conditions": {  
      "yulOptimizer": true  
    }  
  },  
  {  
    "uid": "SOL-2022-3",  
  }
```

(continues on next page)

(continued from previous page)

```

    "name": "DataLocationChangeInInternalOverride",
    "summary": "It was possible to change the data location of the parameters or return variables from ``calldata`` to ``memory`` and vice-versa while overriding internal and public functions. This caused invalid code to be generated when calling such a function internally through virtual function calls.",
    "description": "When calling external functions, it is irrelevant if the data location of the parameters is ``calldata`` or ``memory``, the encoding of the data does not change. Because of that, changing the data location when overriding external functions is allowed. The compiler incorrectly also allowed a change in the data location for overriding public and internal functions. Since public functions can be called internally as well as externally, this causes invalid code to be generated when such an incorrectly overridden function is called internally through the base contract. The caller provides a memory pointer, but the called function interprets it as a calldata pointer or vice-versa.",
    "link": "https://blog.soliditylang.org/2022/05/17/data-location-inheritance-bug/",
    "introduced": "0.6.9",
    "fixed": "0.8.14",
    "severity": "very low"
},
{
    "uid": "SOL-2022-2",
    "name": "NestedCallDataArrayAbiReencodingSizeValidation",
    "summary": "ABI-reencoding of nested dynamic calldata arrays did not always perform proper size checks against the size of calldata and could read beyond ``calldatasize()``.",
    "description": "Calldata validation for nested dynamic types is deferred until the first access to the nested values. Such an access may for example be a copy to memory or an index or member access to the outer type. While in most such accesses calldata validation correctly checks that the data area of the nested array is completely contained in the passed calldata (i.e. in the range [0, calldatasize()]), this check may not be performed, when ABI encoding such nested types again directly from calldata. For instance, this can happen, if a value in calldata with a nested dynamic array is passed to an external call, used in ``abi.encode`` or emitted as event. In such cases, if the data area of the nested array extends beyond ``calldatasize()``, ABI encoding it did not revert, but continued reading values from beyond ``calldatasize()`` (i.e. zero values).",
    "link": "https://blog.soliditylang.org/2022/05/17/calldata-reencode-size-check-bug/",
    "introduced": "0.5.8",
    "fixed": "0.8.14",
    "severity": "very low"
},
{
    "uid": "SOL-2022-1",
    "name": "AbiEncodeCallLiteralAsFixedBytesBug",
    "summary": "Literals used for a fixed length bytes parameter in ``abi.encodeCall`` were encoded incorrectly.",
    "description": "For the encoding, the compiler only considered the types of the expressions in the second argument of ``abi.encodeCall`` itself, but not the parameter types of the function given as first argument. In almost all cases the abi encoding of the type of the expression matches the abi encoding of the parameter type of the given function. This is because the type checker ensures the expression is implicitly convertible to the respective parameter type. However this is not true for number literals used for fixed bytes types shorter than 32 bytes, nor for string literals used for any fixed bytes type. Number literals were encoded as numbers instead of being shifted to become left-aligned. String literals were encoded as dynamically sized memory strings instead of being converted to a left-aligned bytes value."

```

(continued from previous page)

```

"link": "https://blog.soliditylang.org/2022/03/16/encodecall-bug/",
"introduced": "0.8.11",
"fixed": "0.8.13",
"severity": "very low"

},
{
  "uid": "SOL-2021-4",
  "name": "UserDefinedValueTypesBug",
  "summary": "User defined value types with underlying type shorter than 32 bytes used incorrect storage layout and wasted storage",
  "description": "The compiler did not correctly compute the storage layout of user defined value types based on types that are shorter than 32 bytes. It would always use a full storage slot for these types, even if the underlying type was shorter. This was wasteful and might have problems with tooling or contract upgrades.",
  "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-bug/",
  "introduced": "0.8.8",
  "fixed": "0.8.9",
  "severity": "very low"
},
{
  "uid": "SOL-2021-3",
  "name": "SignedImmutables",
  "summary": "Immutable variables of signed integer type shorter than 256 bits can lead to values with invalid higher order bits if inline assembly is used.",
  "description": "When immutable variables of signed integer type shorter than 256 bits are read, their higher order bits were unconditionally set to zero. The correct operation would be to sign-extend the value, i.e. set the higher order bits to one if the sign bit is one. This sign-extension is performed by Solidity just prior to when it matters, i.e. when a value is stored in memory, when it is compared or when a division is performed. Because of that, to our knowledge, the only way to access the value in its unclean state is by reading it through inline assembly.",
  "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
  "introduced": "0.6.5",
  "fixed": "0.8.9",
  "severity": "very low"
},
{
  "uid": "SOL-2021-2",
  "name": "ABIDecodeTwoDimensionalArrayMemory",
  "summary": "If used on memory byte arrays, result of the function ``abi.decode`` can depend on the contents of memory outside of the actual byte array that is decoded.",
  "description": "The ABI specification uses pointers to data areas for everything that is dynamically-sized. When decoding data from memory (instead of calldata), the ABI decoder did not properly validate some of these pointers. More specifically, it was possible to use large values for the pointers inside arrays such that computing the offset resulted in an undetected overflow. This could lead to these pointers targeting areas in memory outside of the actual area to be decoded. This way, it was possible for ``abi.decode`` to return different values for the same encoded byte array.",
  "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/"
}

```

(continues on next page)

(continued from previous page)

```

    "introduced": "0.4.16",
    "fixed": "0.8.4",
    "conditions": {
        "ABIEncoderV2": true
    },
    "severity": "very low"
},
{
    "uid": "SOL-2021-1",
    "name": "KeccakCaching",
    "summary": "The bytecode optimizer incorrectly re-used previously evaluated Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256 hashes in inline assembly.",
    "description": "Solidity's bytecode optimizer has a step that can compute Keccak-256 hashes, if the contents of the memory are known during compilation time. This step also has a mechanism to determine that two Keccak-256 hashes are equal even if the values in memory are not known during compile time. This mechanism had a bug where Keccak-256 of the same memory content, but different sizes were considered equal. More specifically, ``keccak256(mpos1, length1)`` and ``keccak256(mpos2, length2)`` in some cases were considered equal if ``length1`` and ``length2``, when rounded up to nearest multiple of 32 were the same, and when the memory contents at ``mpos1`` and ``mpos2`` can be deduced to be equal. You maybe affected if you compute multiple Keccak-256 hashes of the same content, but with different lengths inside inline assembly. You are unaffected if your code uses ``keccak256`` with a length that is not a compile-time constant or if it is always a multiple of 32.",
    "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
    "fixed": "0.8.3",
    "conditions": {
        "optimizer": true
    },
    "severity": "medium"
},
{
    "uid": "SOL-2020-11",
    "name": "EmptyByteArrayCopy",
    "summary": "Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data.",
    "description": "The routine that copies byte arrays from memory or calldata to storage stores unrelated data from after the source array in the storage slot if the source array is empty. If the storage array's length is subsequently increased either by using ``.push()`` or by assigning to its ``.length`` attribute (only before 0.6.0), the newly created byte array elements will not be zero-initialized, but contain the unrelated data. You are not affected if you do not assign to ``.length`` and do not use ``.push()`` on byte arrays, or only use ``.push(<arg>)`` or manually initialize the new elements.",
    "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
    "fixed": "0.7.4",
    "severity": "medium"
},
{
    "uid": "SOL-2020-10",

```

(continues on next page)

(continued from previous page)

```

    "name": "DynamicArrayCleanup",
    "summary": "When assigning a dynamically-sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out.",
    "description": "Consider a dynamically-sized array in storage whose base-type is small enough such that multiple values can be packed into a single slot, such as `uint128[]`. Let us define its length to be `l`. When this array gets assigned from another array with a smaller length, say `m`, the slots between elements `m` and `l` have to be cleaned by zeroing them out. However, this cleaning was not performed properly. Specifically, after the slot corresponding to `m`, only the first packed value was cleaned up. If this array gets resized to a length larger than `m`, the indices corresponding to the unclean parts of the slot contained the original value, instead of 0. The resizing here is performed by assigning to the array `length`, by a `push()` or via inline assembly. You are not affected if you are only using `push(<arg>)` or if you assign a value (even zero) to the new elements after increasing the length of the array.",
    "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-bug/",
    "fixed": "0.7.3",
    "severity": "medium"
},
{
    "uid": "SOL-2020-9",
    "name": "FreeFunctionRedefinition",
    "summary": "The compiler does not flag an error when two or more free functions with the same name and parameter types are defined in a source unit or when an imported free function alias shadows another free function with a different name but identical parameter types.",
    "description": "In contrast to functions defined inside contracts, free functions with identical names and parameter types did not create an error. Both definition of free functions with identical name and parameter types and an imported free function with an alias that shadows another function with a different name but identical parameter types were permitted due to which a call to either the multiply-defined free function or the imported free function alias within a contract led to the execution of that free function which was defined first within the source unit. Subsequently defined identical free function definitions were silently ignored and their code generation was skipped.",
    "introduced": "0.7.1",
    "fixed": "0.7.2",
    "severity": "low"
},
{
    "uid": "SOL-2020-8",
    "name": "UsingForCalldata",
    "summary": "Function calls to internal library functions with calldata parameters called via ``using for`` can result in invalid data being read.",
    "description": "Function calls to internal library functions using the ``using for`` mechanism copied all calldata parameters to memory first and passed them on like that, regardless of whether it was an internal or an external call. Due to that, the called function would receive a memory pointer that is interpreted as a calldata pointer. Since dynamically sized arrays are passed using two stack slots for calldata, but only one for memory, this can lead to stack corruption. An affected library call will consider the JUMPDEST to which it is supposed to return as part of (continues on next page) and will instead jump out to whatever was on the stack before the call."
}

```

(continued from previous page)

```

    "introduced": "0.6.9",
    "fixed": "0.6.10",
    "severity": "very low"
},
{
    "uid": "SOL-2020-7",
    "name": "MissingEscapingInFormatting",
    "summary": "String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled.",
    "description": "When ABIEncoderV2 is enabled, string literals passed directly to encoding functions or external function calls are stored as strings in the intermediate code. Characters outside the printable range are handled correctly, but backslashes are not escaped in this procedure. This leads to double backslashes being reduced to single backslashes and consequently re-interpreted as escapes potentially resulting in a different string being encoded.",
    "introduced": "0.5.14",
    "fixed": "0.6.8",
    "severity": "very low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2020-6",
    "name": "ArraySliceDynamicallyEncodedBaseType",
    "summary": "Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read.",
    "description": "For arrays with dynamically sized base types, index range accesses that use a start expression that is non-zero will result in invalid array slices. Any index access to such array slices will result in data being read from incorrect calldata offsets. Array slices are only supported for dynamic calldata types and all problematic type require ABIEncoderV2 to be enabled.",
    "introduced": "0.6.0",
    "fixed": "0.6.8",
    "severity": "very low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2020-5",
    "name": "ImplicitConstructorCallvalueCheck",
    "summary": "The creation code of a contract that does not define a constructor but has a base that does define a constructor did not revert for calls with non-zero value.",
    "description": "Starting from Solidity 0.4.5 the creation code of contracts without explicit payable constructor is supposed to contain a callvalue check that results in contract creation reverting, if non-zero value is passed. However, this check was missing in case no explicit constructor was defined in a contract at all, but the contract has a base that does define a constructor. In these cases it is possible to send value in a contract creation transaction or using inline assembly without revert, even though the creation code is supposed to be non-payable."
}

```

(continues on next page)

(continued from previous page)

```

    "introduced": "0.4.5",
    "fixed": "0.6.8",
    "severity": "very low"
},
{
    "uid": "SOL-2020-4",
    "name": "TupleAssignmentMultiStackSlotComponents",
    "summary": "Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values.",
    "description": "Tuple assignments did not correctly account for tuple components that occupy multiple stack slots in case the number of stack slots differs between left-hand-side and right-hand-side. This can either happen in the presence of nested tuples or if the right-hand-side contains external function pointers or references to dynamic calldata arrays, while the left-hand-side contains an omission.",
    "introduced": "0.1.6",
    "fixed": "0.6.6",
    "severity": "very low"
},
{
    "uid": "SOL-2020-3",
    "name": "MemoryArrayCreationOverflow",
    "summary": "The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption.",
    "description": "No runtime overflow checks were performed for the length of memory arrays during creation. In cases for which the memory size of an array in bytes, i.e. the array length times 32, is larger than 2^256-1, the memory allocation will overflow, potentially resulting in overlapping memory areas. The length of the array is still stored correctly, so copying or iterating over such an array will result in out-of-gas.",
    "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/",
    "introduced": "0.2.0",
    "fixed": "0.6.5",
    "severity": "low"
},
{
    "uid": "SOL-2020-1",
    "name": "YulOptimizerRedundantAssignmentBreakContinue",
    "summary": "The Yul optimizer can remove essential assignments to variables declared inside for loops when Yul's continue or break statement is used. You are unlikely to be affected if you do not use inline assembly with for loops and continue and break statements.",
    "description": "The Yul optimizer has a stage that removes assignments to variables that are overwritten again or are not used in all following control-flow branches. This logic incorrectly removes such assignments to variables declared inside a for loop if they can be removed in a control-flow branch that ends with ``break`` or ``continue`` even though they cannot be removed in other control-flow branches. Variables declared outside of the respective for loop are not affected.",
    "introduced": "0.6.0",
    "fixed": "0.6.1",
    "severity": "medium",
    "conditions": {

```

(continues on next page)

(continued from previous page)

```

        "yulOptimizer": true
    }
},
{
    "uid": "SOL-2020-2",
    "name": "privateCanBeOverridden",
    "summary": "Private methods can be overridden by inheriting contracts.",
    "description": "While private methods of base contracts are not visible and\u202a
    cannot be called directly from the derived contract, it is still possible to declare a\u202a
    function of the same name and type and thus change the behaviour of the base contract\u202a
    's function.",
    "introduced": "0.3.0",
    "fixed": "0.5.17",
    "severity": "low"
},
{
    "uid": "SOL-2020-1",
    "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
    "summary": "The Yul optimizer can remove essential assignments to variables\u202a
    declared inside for loops when Yul's continue or break statement is used. You are\u202a
    unlikely to be affected if you do not use inline assembly with for loops and continue\u202a
    and break statements.",
    "description": "The Yul optimizer has a stage that removes assignments to\u202a
    variables that are overwritten again or are not used in all following control-flow\u202a
    branches. This logic incorrectly removes such assignments to variables declared inside\u202a
    a for loop if they can be removed in a control-flow branch that ends with ``break`` or\u202a
    ``continue`` even though they cannot be removed in other control-flow branches.\u202a
    Variables declared outside of the respective for loop are not affected.",
    "introduced": "0.5.8",
    "fixed": "0.5.16",
    "severity": "low",
    "conditions": {
        "yulOptimizer": true
    }
},
{
    "uid": "SOL-2019-10",
    "name": "ABIEncoderV2LoopYulOptimizer",
    "summary": "If both the experimental ABIEncoderV2 and the experimental Yul\u202a
    optimizer are activated, one component of the Yul optimizer may reuse data in memory\u202a
    that has been changed in the meantime.",
    "description": "The Yul optimizer incorrectly replaces ``mload`` and ``sload``\u202a
    calls with values that have been previously written to the load location (and\u202a
    potentially changed in the meantime) if all of the following conditions are met: (1)\u202a
    there is a matching ``mstore`` or ``sstore`` call before; (2) the contents of memory\u202a
    or storage is only changed in a function that is called (directly or indirectly) in\u202a
    between the first store and the load call; (3) called function contains a for loop\u202a
    where the same memory location is changed in the condition or the post or body block.\u202a
    When used in Solidity mode, this can only happen if the experimental ABIEncoderV2 is\u202a
    activated and the experimental Yul optimizer has been activated manually in addition\u202a
    to the regular optimizer in the compiler settings.",
    "introduced": "0.5.14",
}

```

(continues on next page)

(continued from previous page)

```

    "fixed": "0.5.15",
    "severity": "low",
    "conditions": {
        "ABIEncoderV2": true,
        "optimizer": true,
        "yulOptimizer": true
    }
},
{
    "uid": "SOL-2019-9",
    "name": "ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
    "summary": "Reading from calldata structs that contain dynamically encoded, but statically-sized members can result in incorrect values.",
    "description": "When a calldata struct contains a dynamically encoded, but statically-sized member, the offsets for all subsequent struct members are calculated incorrectly. All reads from such members will result in invalid values. Only calldata structs are affected, i.e. this occurs in external functions with such structs as argument. Using affected structs in storage or memory or as arguments to public functions on the other hand works correctly.",
    "introduced": "0.5.6",
    "fixed": "0.5.11",
    "severity": "low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2019-8",
    "name": "SignedArrayStorageCopy",
    "summary": "Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array.",
    "description": "In two's complement, negative integers have their higher order bits set. In order to fit into a shared storage slot, these have to be set to zero. When a conversion is done at the same time, the bits to set to zero were incorrectly determined from the source and not the target type. This means that such copy operations can lead to incorrect values being stored.",
    "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
    "introduced": "0.4.7",
    "fixed": "0.5.10",
    "severity": "low/medium"
},
{
    "uid": "SOL-2019-7",
    "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
    "summary": "Storage arrays containing structs or other statically-sized arrays are not read properly when directly encoded in external function calls or in abi.encode*.",
    "description": "When storage arrays whose elements occupy more than a single storage slot are directly encoded in external function calls or using abi.encode*, their elements are read in an overlapping manner, i.e. the element pointer is not properly advanced between reads. This is not a problem when the storage data is first copied to a memory variable or if the storage array only contains value (continues on next page) dynamically-sized arrays."
}

```

(continued from previous page)

```

"link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
"introduced": "0.4.16",
"fixed": "0.5.10",
"severity": "low",
"conditions": {
    "ABIEncoderV2": true
},
{
    "uid": "SOL-2019-6",
    "name": "DynamicConstructorArgumentsClippedABIV2",
    "summary": "A contract's constructor that takes structs or arrays that contain\u2192 dynamically-sized arrays reverts or decodes to invalid data.",
    "description": "During construction of a contract, constructor parameters are\u2192 copied from the code section to memory for decoding. The amount of bytes to copy was\u2192 calculated incorrectly in case all parameters are statically-sized but contain\u2192 dynamically-sized arrays as struct members or inner arrays. Such types are only\u2192 available if ABIEncoderV2 is activated.",
    "introduced": "0.4.16",
    "fixed": "0.5.9",
    "severity": "very low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2019-5",
    "name": "UninitializedFunctionPointerInConstructor",
    "summary": "Calling uninitialized internal function pointers created in the\u2192 constructor does not always revert and can cause unexpected behaviour.",
    "description": "Uninitialized internal function pointers point to a special\u2192 piece of code that causes a revert when called. Jump target positions are different\u2192 during construction and after deployment, but the code for setting this special jump\u2192 target only considered the situation after deployment.",
    "introduced": "0.5.0",
    "fixed": "0.5.8",
    "severity": "very low"
},
{
    "uid": "SOL-2019-5",
    "name": "UninitializedFunctionPointerInConstructor_0.4.x",
    "summary": "Calling uninitialized internal function pointers created in the\u2192 constructor does not always revert and can cause unexpected behaviour.",
    "description": "Uninitialized internal function pointers point to a special\u2192 piece of code that causes a revert when called. Jump target positions are different\u2192 during construction and after deployment, but the code for setting this special jump\u2192 target only considered the situation after deployment.",
    "introduced": "0.4.5",
    "fixed": "0.4.26",
    "severity": "very low"
},
{

```

(continues on next page)

(continued from previous page)

```

    "uid": "SOL-2019-4",
    "name": "IncorrectEventSignatureInLibraries",
    "summary": "Contract types used in events in libraries cause an incorrect event\u202a\u202a signature hash",
    "description": "Instead of using the type `address` in the hashed signature, the\u202a\u202a actual contract name was used, leading to a wrong hash in the logs.",
    "introduced": "0.5.0",
    "fixed": "0.5.8",
    "severity": "very low"
},
{
    "uid": "SOL-2019-4",
    "name": "IncorrectEventSignatureInLibraries_0.4.x",
    "summary": "Contract types used in events in libraries cause an incorrect event\u202a\u202a signature hash",
    "description": "Instead of using the type `address` in the hashed signature, the\u202a\u202a actual contract name was used, leading to a wrong hash in the logs.",
    "introduced": "0.3.0",
    "fixed": "0.4.26",
    "severity": "very low"
},
{
    "uid": "SOL-2019-3",
    "name": "ABIEncoderV2PackedStorage",
    "summary": "Storage structs and arrays with types shorter than 32 bytes can\u202a\u202a cause data corruption if encoded directly from storage using the experimental\u202a\u202a ABIEncoderV2.",
    "description": "Elements of structs and arrays that are shorter than 32 bytes\u202a\u202a are not properly decoded from storage when encoded directly (i.e. not via a memory\u202a\u202a type) using ABIEncoderV2. This can cause corruption in the values themselves but can\u202a\u202a also overwrite other parts of the encoded data.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-\u202a\u202a abiencoderv2-bug/",
    "introduced": "0.5.0",
    "fixed": "0.5.7",
    "severity": "low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2019-3",
    "name": "ABIEncoderV2PackedStorage_0.4.x",
    "summary": "Storage structs and arrays with types shorter than 32 bytes can\u202a\u202a cause data corruption if encoded directly from storage using the experimental\u202a\u202a ABIEncoderV2.",
    "description": "Elements of structs and arrays that are shorter than 32 bytes\u202a\u202a are not properly decoded from storage when encoded directly (i.e. not via a memory\u202a\u202a type) using ABIEncoderV2. This can cause corruption in the values themselves but can\u202a\u202a also overwrite other parts of the encoded data.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-\u202a\u202a abiencoderv2-bug/",
}

```

(continues on next page)

(continued from previous page)

```

    "introduced": "0.4.19",
    "fixed": "0.4.26",
    "severity": "low",
    "conditions": {
        "ABIEncoderV2": true
    }
},
{
    "uid": "SOL-2019-2",
    "name": "IncorrectByteInstructionOptimization",
    "summary": "The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values.",
    "description": "The optimizer incorrectly handles byte opcodes that use the constant 31 as second argument. This can happen when performing index access on bytesNN types with a compile-time constant value (not index) of 31 or when using the byte opcode in inline assembly.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-abiencoderv2-bug/",
    "introduced": "0.5.5",
    "fixed": "0.5.7",
    "severity": "very low",
    "conditions": {
        "optimizer": true
    }
},
{
    "uid": "SOL-2019-1",
    "name": "DoubleShiftSizeOverflow",
    "summary": "Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values.",
    "description": "Nested logical shift operations whose total shift size is 2**256 or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-abiencoderv2-bug/",
    "introduced": "0.5.5",
    "fixed": "0.5.6",
    "severity": "low",
    "conditions": {
        "optimizer": true,
        "evmVersion": ">=constantinople"
    }
},
{
    "uid": "SOL-2018-4",
    "name": "ExpExponentCleanup",
    "summary": "Using the ** operator with an exponent of type shorter than 256 bits can result in unexpected values.",
    "description": "Higher order bits in the exponent are not properly cleaned before the EXP opcode is applied if the type of the exponent expression is smaller than 256 bits and not smaller than the type of the base. In that case, the result might be larger than expected if the exponent is assumed to lie within the full range of the type. Literal numbers as exponents are unaffected as are exponents or bases of type uint256."
}

```

(continues on next page)

(continued from previous page)

```

"link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
"fixed": "0.4.25",
"severity": "medium/high",
"check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
},
{
  "uid": "SOL-2018-3",
  "name": "EventStructWrongData",
  "summary": "Using structs in events logged wrong data.",
  "description": "If a struct is used in an event, the address of the struct is\u202alogged instead of the actual data.",
  "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
  "introduced": "0.4.17",
  "fixed": "0.4.25",
  "severity": "very low",
  "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')].[?(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith('struct'))]"}
},
{
  "uid": "SOL-2018-2",
  "name": "NestedArrayFunctionCallDecoder",
  "summary": "Calling functions that return multi-dimensional fixed-size arrays\u202acan result in memory corruption.",
  "description": "If Solidity code calls a function that returns a multi-dimensional fixed-size array, array elements are incorrectly interpreted as memory\u202apointers and thus can cause memory corruption if the return values are accessed.\u202aCalling functions with multi-dimensional fixed-size arrays is unaffected as is.\u202areturning fixed-size arrays from function calls. The regular expression only checks if such functions are present, not if they are called, which is required for the contract to be affected.",
  "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
  "introduced": "0.1.4",
  "fixed": "0.4.22",
  "severity": "medium",
  "check": {"regex-source": "returns[^;]*\\[\\s*[^\n\f][\n\f]*\\] \\t\\r\\n\\v\\f[^;]*\\]\\s*\\[\\s*[^\n\f][\n\f]*\\] \\t\\r\\n\\v\\f[^;]*\\][^;]*[;]"}
},
{
  "uid": "SOL-2018-1",
  "name": "OneOfTwoConstructorsSkipped",
  "summary": "If a contract has both a new-style constructor (using the\u202aconstructor keyword) and an old-style constructor (a function with the same name as\u202athe contract) at the same time, one of them will be ignored.",
  "description": "If a contract has both a new-style constructor (using the\u202aconstructor keyword) and an old-style constructor (a function with the same name as\u202athe contract) at the same time, one of them will be ignored. There will be a compiler\u202awarning about the old-style constructor, so contracts only using new-style\u202aconstructors are fine.",
  "introduced": "0.4.22",
  "fixed": "0.4.23",
  "severity": "very low"
}

```

(continues on next page)

(continued from previous page)

```

},
{
  "uid": "SOL-2017-5",
  "name": "ZeroFunctionSelector",
  "summary": "It is possible to craft the name of a function such that it is ↵
executed instead of the fallback function in very specific circumstances.",
  "description": "If a function has a selector consisting only of zeros, is ↵
payable and part of a contract that does not have a fallback function and at most five ↵
external functions in total, this function is called instead of the fallback function ↵
if Ether is sent to the contract without data.",
  "fixed": "0.4.18",
  "severity": "very low"
},
{
  "uid": "SOL-2017-4",
  "name": "DelegateCallReturnValue",
  "summary": "The low-level .delegatecall() does not return the execution outcome, ↵
but converts the value returned by the functioned called to a boolean instead.",
  "description": "The return value of the low-level .delegatecall() function is ↵
taken from a position in memory, where the call data or the return data resides. This ↵
value is interpreted as a boolean and put onto the stack. This means if the called ↵
function returns at least 32 zero bytes, .delegatecall() returns false even if the ↵
call was successful.",
  "introduced": "0.3.0",
  "fixed": "0.4.15",
  "severity": "low"
},
{
  "uid": "SOL-2017-3",
  "name": "ECRecoverMalformedInput",
  "summary": "The ecrecover() builtin can return garbage for malformed input.",
  "description": "The ecrecover precompile does not properly signal failure for ↵
malformed input (especially in the 'v' argument) and thus the Solidity function can ↵
return data that was previously present in the return area in memory.",
  "fixed": "0.4.14",
  "severity": "medium"
},
{
  "uid": "SOL-2017-2",
  "name": "SkipEmptyStringLiteral",
  "summary": "If \"\" is used in a function call, the following function arguments ↵
will not be correctly passed to the function.",
  "description": "If the empty string literal \"\" is used as an argument in a ↵
function call, it is skipped by the encoder. This has the effect that the encoding of ↵
all arguments following this is shifted left by 32 bytes and thus the function call ↵
data is corrupted.",
  "fixed": "0.4.12",
  "severity": "low"
},
{
  "uid": "SOL-2017-1",
  "name": "ConstantOptimizerSubtraction",

```

(continues on next page)

(continued from previous page)

```

    "summary": "In some situations, the optimizer replaces certain numbers in the code with routines that compute different numbers.",
    "description": "The optimizer tries to represent any number in the bytecode by routines that compute them with less gas. For some special numbers, an incorrect routine is generated. This could allow an attacker to e.g. trick victims about a specific amount of ether, or function calls to call different functions (or none at all).",
    "link": "https://blog.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
    "fixed": "0.4.11",
    "severity": "low",
    "conditions": {
        "optimizer": true
    },
    {
        "uid": "SOL-2016-11",
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying memory, ignored its return value. On the public chain, calls to the identity precompile can be made in a way that they never fail, but this might be different on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "uid": "SOL-2016-10",
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At jump destinations, multiple code paths join and thus it has to compute a common state from the incoming edges. Computing this common state was simplified to just use the empty state, but this implementation was not done properly. This bug can cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-9",
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same 32-byte storage slot, but storage writes always write 32 bytes. For some types, the higher order bytes were not cleaned properly, which made it sometimes possible to overwrite a variable in storage when writing to another one.",
        "link": "https://blog.soliditylang.org/2016/11/01/security-alert-solidity-variables-can-overwritten-storage/"
    }
}

```

(continues on next page)

(continued from previous page)

```

    "severity": "high",
    "introduced": "0.1.6",
    "fixed": "0.4.4"
},
{
    "uid": "SOL-2016-8",
    "name": "OptimizerStaleKnowledgeAboutSHA3",
    "summary": "The optimizer did not properly reset its knowledge about SHA3\u2014operations resulting in some hashes (also used for storage variable positions) not\u2014being calculated correctly.",
    "description": "The optimizer performs symbolic execution in order to save re-\u2014evaluating expressions whose value is already known. This knowledge was not properly\u2014reset across control flow paths and thus the optimizer sometimes thought that the\u2014result of a SHA3 operation is already present on the stack. This could result in data\u2014corruption by accessing the wrong storage slot.",
    "severity": "medium",
    "fixed": "0.4.3",
    "conditions": {
        "optimizer": true
    }
},
{
    "uid": "SOL-2016-7",
    "name": "LibrariesNotCallableFromPayableFunctions",
    "summary": "Library functions threw an exception when called from a call that\u2014received Ether.",
    "description": "Library functions are protected against sending them Ether\u2014through a call. Since the DELEGATECALL opcode forwards the information about how much\u2014Ether was sent with a call, the library function incorrectly assumed that Ether was\u2014sent to the library and threw an exception.",
    "severity": "low",
    "introduced": "0.4.0",
    "fixed": "0.4.2"
},
{
    "uid": "SOL-2016-6",
    "name": "SendFailsForZeroEther",
    "summary": "The send function did not provide enough gas to the recipient if no\u2014Ether was sent with it.",
    "description": "The recipient of an Ether transfer automatically receives a\u2014certain amount of gas from the EVM to handle the transfer. In the case of a zero-\u2014transfer, this gas is not provided which causes the recipient to throw an exception.",
    "severity": "low",
    "fixed": "0.4.0"
},
{
    "uid": "SOL-2016-5",
    "name": "DynamicAllocationInfiniteLoop",
    "summary": "Dynamic allocation of an empty memory array caused an infinite loop\u2014and thus an exception.",
    "description": "Memory arrays can be created provided a length. If this length\u2014is zero, code was generated that did not terminate and thus consumed all gas."
}

```

(continues on next page)

(continued from previous page)

```

        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "uid": "SOL-2016-4",
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump\u202a destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At\u202a jump destinations, multiple code paths join and thus it has to compute a common state\u202a from the incoming edges. Computing this common state was not done correctly. This bug\u202a can cause data corruption, but it is probably quite hard to use for targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-3",
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned before\u202a comparison.",
        "description": "Two variables of type bytesNN were considered different if their\u202a higher order bits, which are not part of the actual value, were different. An attacker\u202a might use this to reach seemingly unreachable code paths by providing incorrectly\u202a formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "uid": "SOL-2016-2",
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32 bytes\u202a did not correctly clean the higher order bits, causing corruption in other array\u202a elements.",
        "description": "Multiple elements of an array of values that are shorter than 17\u202a bytes are packed into the same storage slot. Writing to a single element of such an\u202a array did not properly clean the higher order bytes and thus could lead to data\u202a corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "uid": "SOL-2016-1",
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several\u202a undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting\u202a from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
}

```

(continues on next page)

(continued from previous page)

}
]

3.36 Contributing

Help is always welcome and there are plenty of options how you can contribute to Solidity.

In particular, we appreciate support in the following areas:

- Reporting issues.
- Fixing and responding to [Solidity's GitHub issues](#), especially those tagged as “good first issue” which are meant as introductory issues for external contributors.
- Improving the documentation.
- Translating the documentation into more languages.
- Responding to questions from other users on [StackExchange](#) and the [Solidity Gitter Chat](#).
- Getting involved in the language design process by proposing language changes or new features in the [Solidity forum](#) and providing feedback.

To get started, you can try [Building from Source](#) in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

Please note that this project is released with a [Contributor Code of Conduct](#). By participating in this project - in the issues, pull requests, or Gitter channels - you agree to abide by its terms.

3.36.1 Team Calls

If you have issues or pull requests to discuss, or are interested in hearing what the team and contributors are working on, you can join our public team calls:

- Mondays at 3pm CET/CEST.
- Wednesdays at 2pm CET/CEST.

Both calls take place on [Jitsi](#).

3.36.2 How to Report Issues

To report an issue, please use the [GitHub issues tracker](#). When reporting issues, please mention the following details:

- Solidity version.
- Source code (if applicable).
- Operating system.
- Steps to reproduce the issue.
- Actual vs. expected behaviour.

Reducing the source code that caused the issue to a bare minimum is always very helpful and sometimes even clarifies a misunderstanding.

3.36.3 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the [Solidity Development Gitter channel](#) (different from the one mentioned above, this one is focused on compiler and language development instead of language usage) first.

New features and bugfixes should be added to the `Changelog.md` file: please follow the style of previous entries, when applicable.

Finally, please make sure you respect the [coding style](#) for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

Thank you for your help!

3.36.4 Running the Compiler Tests

Prerequisites

For running all compiler tests you may want to optionally install a few dependencies (`evmone`, `libz3`, and `libhera`).

On macOS some of the testing scripts expect GNU coreutils to be installed. This can be easiest accomplished using Homebrew: `brew install coreutils`.

On Windows systems make sure that you have a privilege to create symlinks, otherwise several tests may fail. Administrators should have that privilege, but you may also [grant it to other users](#) or [enable Developer Mode](#).

Running the Tests

Solidity includes different types of tests, most of them bundled into the [Boost C++ Test Framework](#) application `soltest`. Running `build/test/soltest` or its wrapper `scripts/soltest.sh` is sufficient for most changes.

The `./scripts/tests.sh` script executes most Solidity tests automatically, including those bundled into the [Boost C++ Test Framework](#) application `soltest` (or its wrapper `scripts/soltest.sh`), as well as command line tests and compilation tests.

The test system automatically tries to discover the location of the `evmone` for running the semantic tests.

The `evmone` library must be located in the `deps` or `deps/lib` directory relative to the current working directory, to its parent or its parent's parent. Alternatively an explicit location for the `evmone` shared object can be specified via the `ETH_EVMONE` environment variable.

`evmone` is needed mainly for running semantic and gas tests. If you do not have it installed, you can skip these tests by passing the `--no-semantic-tests` flag to `scripts/soltest.sh`.

Running Ewasm tests is disabled by default and can be explicitly enabled via `./scripts/soltest.sh --ewasm` and requires `hera` to be found by `soltest`. The mechanism for locating the `hera` library is the same as for `evmone`, except that the variable for specifying an explicit location is called `ETH_HERA`.

The `evmone` and `hera` libraries should both end with the file name extension `.so` on Linux, `.dll` on Windows systems and `.dylib` on macOS.

For running SMT tests, the `libz3` library must be installed and locatable by `cmake` during compiler configure stage.

If the `libz3` library is not installed on your system, you should disable the SMT tests by exporting `SMT_FLAGS=--no-smt` before running `./scripts/tests.sh` or running `./scripts/soltest.sh --no-smt`. These tests are `libsolidity/smtCheckerTests` and `libsolidity/smtCheckerTestsJSON`.

Note: To get a list of all unit tests run by Soltest, run `./build/test/soltest --list_content=HRF`.

For quicker results you can run a subset of, or specific tests.

To run a subset of tests, you can use filters: `./scripts/soltest.sh -t TestSuite/TestName`, where `TestName` can be a wildcard `*`.

Or, for example, to run all the tests for the `yul` disambiguator: `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` has extensive help on all of the options available.

See especially:

- `show_progress (-p)` to show test completion,
- `run_test (-t)` to run specific tests cases, and
- `report-level (-r)` give a more detailed report.

Note: Those working in a Windows environment wanting to run the above basic sets without `libz3`. Using Git Bash, you use: `./build/test/Release/soltest.exe -- --no-smt`. If you are running this in plain Command Prompt, use `.\build\test\Release\soltest.exe -- --no-smt`.

If you want to debug using GDB, make sure you build differently than the “usual”. For example, you could run the following command in your build folder: .. code-block:: bash

```
cmake -DCMAKE_BUILD_TYPE=Debug .. make
```

This creates symbols so that when you debug a test using the `--debug` flag, you have access to functions and variables in which you can break or print with.

The CI runs additional tests (including `solc-j`s and testing third party Solidity frameworks) that require compiling the Emscripten target.

Writing and Running Syntax Tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside the `tests/libsolidity/syntaxTests` folder. These files must contain annotations, stating the expected result(s) of the respective test. The test suite compiles and checks them against the given expectations.

For example: `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator // ----. The comments that follow the separator are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. If you want the contract to compile without any errors or warning you can leave out the separator and the comments that follow it.

In the above example, the state variable `variable` was declared twice, which is not allowed. This results in a `DeclarationError` stating that the identifier was already declared.

The `isoltest` tool is used for these tests and you can find it under `./build/test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable`:

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./build/test/tools/isoltest` again results in a test failure:

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
Contract:
contract test {
    uint256 variable;
}

Expected result:
DeclarationError: (36-52): Identifier already declared.
Obtained result:
Success
```

`isoltest` prints the expected result next to the obtained result, and also provides a way to edit, update or skip the current contract file, or quit the application.

It offers several options for failing tests:

- `edit`: `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in that order).
- `update`: Updates the expectations for contract under test. This updates the annotations by removing unmet expectations and adding missing expectations. The test is then run again.
- `skip`: Skips the execution of this particular test.
- `quit`: Quits `isoltest`.

All of these options apply to the current contract, except `quit` which stops the entire testing process.

Automatically updating the test above changes it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It now passes again:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note: Choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

3.36.5 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and run a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g., the code contains an error. This way, fuzzing tools can find internal problems in the compiler.

We mainly use [AFL](#) for fuzzing. You need to download and install the AFL packages from your repositories (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler:

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage you should be able to see a message similar to the following:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the `cmake` flags pointing to AFL's clang binaries:

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Otherwise, upon execution the fuzzer halts with an error saying binary is not instrumented:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
compile-time instrumentation to isolate interesting test cases while
mutating the input data. For more information, and for tips on how to
instrument binaries, please see /usr/share/doc/afl-doc/docs/README.
```

(continues on next page)

(continued from previous page)

When source code is not available, you may be able to leverage QEMU mode support. Consult the README for tips on how to enable this. (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer. For that, you can use the -n option - but expect much worse results.)

```
[+] PROGRAM ABORT : No instrumentation detected
    Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This makes it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests:

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
  ↳ SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of. There is also a tool called afl-cmin that can trim input files that result in similar behaviour of the binary.

Now run the fuzzer (the -m extends the size of memory to 60 MB):

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer creates source files that lead to failures in /tmp/fuzzer_reports. Often it finds many similar source files that produce the same error. You can use the tool scripts/uniqueErrors.sh to filter out the unique errors.

3.36.6 Whiskers

Whiskers is a string templating system similar to [Mustache](#). It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache. The template markers {{ and }} are replaced by < and > in order to aid parsing and avoid conflicts with [Yul](#) (The symbols < and > are invalid in inline assembly, while { and } are used to delimit blocks). Another limitation is that lists are only resolved one depth and they do not recurse. This may change in the future.

A rough specification is the following:

Any occurrence of <name> is replaced by the string-value of the supplied variable name without any escaping and without iterated replacements. An area can be delimited by <#name>...</name>. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any <inner> items by their respective value. Top-level variables can also be used inside such areas.

There are also conditionals of the form <?name>...<!name>...</name>, where template replacements continue recursively either in the first or the second segment depending on the value of the boolean parameter name. If <?+name>...<!+name>...</+name> is used, then the check is whether the string parameter name is non-empty.

3.36.7 Documentation Style Guide

In the following section you find style recommendations specifically focusing on documentation contributions to Solidity.

English Language

Use English, with British English spelling preferred, unless using project or brand names. Try to reduce the usage of local slang and references, making your language as clear to all readers as possible. Below are some references to help:

- Simplified technical English
- International English
- British English spelling

Note: While the official Solidity documentation is written in English, there are community contributed [Translations](#) in other languages available. Please refer to the [translation guide](#) for information on how to contribute to the community translations.

Title Case for Headings

Use title case for headings. This means capitalise all principal words in titles, but not articles, conjunctions, and prepositions unless they start the title.

For example, the following are all correct:

- Title Case for Headings.
- For Headings Use Title Case.
- Local and State Variable Names.
- Order of Layout.

Expand Contractions

Use expanded contractions for words, for example:

- “Do not” instead of “Don’t”.
- “Can not” instead of “Can’t”.

Active and Passive Voice

Active voice is typically recommended for tutorial style documentation as it helps the reader understand who or what is performing a task. However, as the Solidity documentation is a mixture of tutorials and reference content, passive voice is sometimes more applicable.

As a summary:

- Use passive voice for technical reference, for example language definition and internals of the Ethereum VM.
- Use active voice when describing recommendations on how to apply an aspect of Solidity.

For example, the below is in passive voice as it specifies an aspect of Solidity:

Functions can be declared `pure` in which case they promise not to read from or modify the state.

For example, the below is in active voice as it discusses an application of Solidity:

When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings.

Common Terms

- “Function parameters” and “return variables”, not input and output parameters.

Code Examples

A CI process tests all code block formatted code examples that begin with `pragma solidity, contract, library` or `interface` using the `./test/cmdlineTests.sh` script when you create a PR. If you are adding new code examples, ensure they work and pass tests before creating the PR.

Ensure that all code examples begin with a `pragma` version that spans the largest where the contract code is valid. For example `pragma solidity >=0.4.0 <0.9.0;`.

Running Documentation Tests

Make sure your contributions pass our documentation tests by running `./docs/docs.sh` that installs dependencies needed for documentation and checks for any problems such as broken links or syntax issues.

3.36.8 Solidity Language Design

To actively get involved in the language design process and share your ideas concerning the future of Solidity, please join the [Solidity forum](#).

The Solidity forum serves as the place to propose and discuss new language features and their implementation in the early stages of ideation or modifications of existing features.

As soon as proposals get more tangible, their implementation will also be discussed in the [Solidity GitHub repository](#) in the form of issues.

In addition to the forum and issue discussions, we regularly host language design discussion calls in which selected topics, issues or feature implementations are debated in detail. The invitation to those calls is shared via the forum.

We are also sharing feedback surveys and other content that is relevant to language design in the forum.

If you want to know where the team is standing in terms of implementing new features, you can follow the implementation status in the [Solidity Github project](#). Issues in the design backlog need further specification and will either be discussed in a language design call or in a regular team call. You can see the upcoming changes for the next breaking release by changing from the default branch (`develop`) to the `breaking` branch.

For ad-hoc cases and questions you can reach out to us via the [Solidity-dev Gitter channel](#), a dedicated chatroom for conversations around the Solidity compiler and language development.

We are happy to hear your thoughts on how we can improve the language design process to be even more collaborative and transparent.

3.37 Solidity Brand Guide

This brand guide features information on Solidity's brand policy and logo usage guidelines.

3.37.1 The Solidity Brand

The Solidity programming language is an open-source, community project governed by a core team. The core team is sponsored by the [Ethereum Foundation](#).

This document aims to provide information about how to best use the Solidity brand name and logo.

We encourage you to read this document carefully before using the brand name or the logo. Your cooperation is highly appreciated!

3.37.2 Solidity Brand Name

“Solidity” should be used to refer to the Solidity programming language solely.

Please do not use “Solidity”:

- To refer to any other programming language.
- In a way that is misleading or may imply association of unrelated modules, tools, documentation, or other resources with the Solidity programming language.
- In ways that confuse the community as to whether the Solidity programming language is open-source and free to use.

3.37.3 Solidity Logo License



The Solidity logo is distributed and licensed under a [Creative Commons Attribution 4.0 International License](#).

This is the most permissive Creative Commons license and allows reuse and modifications for any purpose.

You are free to:

- **Share** — Copy and redistribute the material in any medium or format.
- **Adapt** — Remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests that the Solidity core team endorses you or your use.

When using the Solidity logo, please respect the Solidity logo guidelines.

3.37.4 Solidity Logo Guidelines

(Right click on the logo to download it.)

Please do not:

- Change the ratio of the logo (do not stretch it or cut it).
- Change the colors of the logo, unless it is absolutely necessary.

3.37.5 Credits

This document was, in parts, derived from the [Python Software Foundation Trademark Usage Policy](#) and the [Rust Media Guide](#).

3.38 Language Influences

Solidity is a curly-bracket language that has been influenced and inspired by several well-known programming languages.

Solidity is most profoundly influenced by C++, but also borrowed concepts from languages like Python, JavaScript, and others.

The influence from C++ can be seen in the syntax for variable declarations, for loops, the concept of overloading functions, implicit and explicit type conversions and many other details.

In the early days of the language, Solidity used to be partly influenced by JavaScript. This was due to function-level scoping of variables and the use of the keyword `var`. The JavaScript influence was reduced starting from version 0.4.0. Now, the main remaining similarity to JavaScript is that functions are defined using the keyword `function`. Solidity also supports import syntax and semantics that are similar to those available in JavaScript. Besides those points, Solidity looks like most other curly-bracket languages and has no major JavaScript influence anymore.

Another influence to Solidity was Python. Solidity's modifiers were added trying to model Python's decorators with a much more restricted functionality. Furthermore, multiple inheritance, C3 linearization, and the `super` keyword are taken from Python as well as the general assignment and copy semantics of value and reference types.

INDEX

Symbols

--allow-paths, 157, 278
--base-path, 157, 158, 276
--include-path, 157, 276
--libraries, 158
--link, 158
--standard-json, 158, 160
<stdin>, 274

A

abi, 89, 90, 214
abstract contract, 136, 138
access
 restricting, 327
account, 11
addmod, 91, 154
address, 11, 55, 59
allowed paths, 157, 278
analyse, 172
anonymous, 156
application binary interface, 214
array, 67, 68, 118
 allocating, 70
 dangling storage references, 74
 length, 72
 literals, 70
 pop, 72
 push, 72
 slice, 76
array of strings, 118
asm, 148, 172, 281
assembly, 148, 281
assert, 91, 103, 154
assignment, 83, 99
 destructuring, 99
auction
 blind, 26
 open, 26

B

balance, 11, 55, 92, 154
ballot, 23

base
 constructor, 136
base class, 129
base path, 157, 276
blind auction, 26
block, 11, 89, 154
 number, 89, 154
 timestamp, 89, 154
bool, 52
break, 94
Bugs, 331
byte array, 58
bytes, 61, 69
bytes members, 90
bytes32, 58
bytes-concat, 69

C

C3 linearization, 137
call, 55, 92
callcode, 13, 92, 140
cast, 85
checked, 101
cleanup, 180
codehash, 92, 154
coding style, 304
coin, 10
coinbase, 89, 154
commandline compiler, 157
comment, 48
common subexpression elimination, 190
compile target, 158
compiler
 commandline, 157
compound operators, 83
constant, 115, 156
constant propagation, 190
constructor, 108, 135
 arguments, 108
continue, 94
contract, 48, 108
 abstract, 136, 138

base, 129
creation, 108
interface, 139
modular, 44
precompiled, 14
contract creation, 14
contract type, 58
contract verification, 210
contracts
 creating, 97
creationCode, 94
cryptography, 91, 154
custom type, 62

D

data, 89, 154
days, 88
deactivate, 14
declarations, 100
default value, 100
delegatecall, 13, 55, 92, 140
delete, 83
deriving, 129
difficulty, 89, 154
direct import, 274
dirty bits, 180
do/while, 94
dynamic array, 118

E

ecrecover, 91, 154
else, 94
encode, 89
encoding, 90
enum, 48, 61
error, 128, 222
errors, 103
escrow, 33
ether, 88
ethereum virtual machine, 11
evaluation order
 expression, 178
 function arguments, 179
event, 9, 48, 125
 anonymous, 125
 indexed, 125
 topic, 125
evm, 11
EVM version, 158
evmasm, 148, 281
exception, 103
experimental, 46
external, 110, 156

F

fallback function, 122
false, 52
file://, 281
filesystem path, 48
finney, 88
fixed, 54
fixed point number, 54
for, 94
function, 48
 call, 13, 94
 external, 94
 fallback, 122
 getter, 112
 internal, 94
 modifier, 48, 113, 327, 329
 pure, 120
 receive ! receive, 121
 view, 119
function parameter, 94
function pointers, 180
function type, 63
functions, 117

G

gas, 12, 89, 154
gas price, 12, 89, 154
getter
 function, 112
goto, 94
gwei, 88

H

Host Filesystem Loader, 272
hours, 88

I

if, 94
import, 47
 direct, 274
 path, 48, 274
 relative, 275
 remapping, 279
import callback, 48, 272
include paths, 157, 276
indexed, 156
inheritance, 129
 multiple, 137
inheritance list, 136
inline
 arrays, 70
installing, 15
instruction, 13

int, 52
 integer, 52
 interface contract, 139
 internal, 110, 156
 iterable mappings, 80
 iulia, 281

J
 julia, 281

K
 keccak256, 91, 154

L
 length, 72
 library, 13, 140, 145
 license, 45
 linearization, 137
 linker, 158
 literal, 59–61
 address, 59
 rational, 59
 string, 60
 location, 67
 log, 14
 lvalue, 83

M
 mapping, 9, 78, 181
 memory, 12, 67
 message call, 13
 metadata, 210
 minutes, 88
 modifiers, 156
 modular contract, 44
 module, 47
 msg, 89, 154
 mulmod, 91, 154

N
 natspec, 48
 new, 70, 97
 number, 89, 154

O
 open auction, 26
 operator, 82
 precedence, 84, 153
 optimiser, 190
 optimizer, 190
 origin, 89, 154
 overload, 124
 overriding

function, 132
 modifier, 134

P
 packed, 90
 parameter, 94
 function, 94
 input, 94
 output, 94
 payable, 156
 pop, 72
 pragma, 45, 46
 precompiled contracts, 14
 precompiles, 14
 private, 110, 156
 public, 110, 156
 purchase, 33
 pure, 156
 pure function, 120
 push, 72

R
 receive ether function, 121
 reference type, 67
 relative import, 275
 remapping
 context, 279
 import, 279
 prefix, 279
 target, 278, 279
 Remix IDE, 48, 281
 remote purchase, 33
 require, 91, 103, 154
 return, 94
 return array, 118
 return string, 118
 return struct, 118
 return variable, 94
 revert, 91, 103, 128, 154
 ripemd160, 91, 154
 runtimeCode, 94

S
 safe math, 101
 safemath, 101
 scoping, 100
 seconds, 88
 selector
 of a function, 149, 214
 of a library function, 144
 of an error, 128, 222
 of an event, 126
 selfdestruct, 14, 93, 154
 send, 55, 92, 154

sender, 89, 154
set, 141
sha256, 91, 154
solc, 157
source file, 47
source mappings, 189
source unit, 47
source unit name, 48, 272
spdx, 45
stack, 12
standard input, 274
standard JSON, 160, 273
state machine, 329
state variable, 48, 181
staticcall, 55, 92
stdin, 274
storage, 11, 12, 67, 181
string, 60, 69, 118
string members, 91
string-concat, 69
struct, 48, 67, 77, 118
style, 304
subcurrency, 8
super, 154
switch, 94
szabo, 88

T

this, 93, 154
throw, 103
time, 88
timestamp, 89, 154
transaction, 10, 12
transfer, 55, 92
true, 52
type, 51, 94
 contract, 58
 conversion, 85
 function, 63
 reference, 67
 struct, 77
 value, 51

U

ufixed, 54
uint, 52
unchecked, 101
user defined value type, 62
using for, 141, 145

V

value, 89, 154
value type, 51
variable

 return, 94
variably sized array, 118
version, 46
VFS, 272
view, 156
view function, 119
virtual filesystem, 48, 272
visibility, 110, 156
voting, 23

W

weeks, 88
wei, 88
while, 94
withdrawal, 325

Y

years, 88
yul, 281

DeFi and the Future of Finance*

Campbell R. Harvey

Duke University, Durham, NC USA 27708

National Bureau of Economic Research, Cambridge MA USA 02138

Ashwin Ramachandran

Dragonfly Capital

Joey Santoro

Fei Protocol

ABSTRACT

Our legacy financial infrastructure has both limited growth opportunities and contributed to the inequality of opportunities. Around the world, 1.7 billion are unbanked. Small businesses, even those with a banking relationship, often must rely on high-cost financing, such as credit cards, because traditional banking excludes them from loan financing. High costs also impact retailers who lose 3% on every credit card sales transaction. These total costs for small businesses are enormous by any metric. The result is less investment and decreased economic growth. Decentralized finance, or DeFi, poses a challenge to the current system and offers a number of potential solutions to the problems inherent in the traditional financial infrastructure. While there are many fintech initiatives, we argue that the ones that embrace the current banking infrastructure are likely to be fleeting. We argue those initiatives that use decentralized methods - in particular blockchain technology - have the best chance to define the future of finance.

Comments: Please comment directly on the live version of our paper. It is available [here](#).

Keywords: Decentralized finance, DeFi; Fintech, Flash loans, Flash swaps, Automatic Market Maker, DEX, Decentralized Exchange, Cryptocurrency, Uniswap, MakerDAO, Compound, Ethereum, Aave, Yield protocol, ERC-20, Initial DeFi Offering, dYdX, Synthetix, Keeper, Set protocol, Yield farming.

JEL: A10, B10, D40, E44, F30, F60, G10, G21, G23, G51, I10, K10, L14, M10, O16, O33, O40, P10, C63, C70, D83, D85

*Current version: April 5, 2021. We appreciate the comments of Dan Robinson, Stani Kulechov, John Mattox, Andreas Park, Chen Feng, Can Gurel, Jeffrey Hoopes, Brian Bernert, Marc Toledo, Marcel Smeets, Ron Nicol, and Daniel Liebau on an earlier draft. Lucy Pless created the graphics and Kay Jaitly provided editorial assistance.

Table of Contents

1. Introduction	4
2. The Origins of Modern Decentralized Finance	8
2.1 A Brief History of Finance	8
2.2 Fintech	8
2.3 Bitcoin and Cryptocurrency	10
2.4 Ethereum and DeFi	12
3. DeFi Infrastructure	13
3.1 Blockchain	13
3.2 Cryptocurrency	14
3.3 The Smart Contract Platform	14
3.4 Oracles	15
3.5 Stablecoins	16
3.6 Decentralized Applications	17
4. DeFi Primitives	18
4.1 Transactions	18
4.2 Fungible Tokens	19
4.2.1 Equity Token	20
4.2.2 Utility Tokens	20
4.2.3 Governance Tokens	21
4.3 Nonfungible Tokens	22
4.3.1 NFT Standard	22
4.3.2 Multi-Token Standard	22
4.4 Custody	23
4.5 Supply Adjustment	23
4.5.1 Burn - Reduce Supply	23
4.5.2 Mint - Increase Supply	24
4.5.3. Bonding Curve - Pricing Supply	24
4.6 Incentives	27
4.6.1 Staking Rewards	27
4.6.2 Slashing (Staking Penalties)	28
4.6.3 Direct Rewards and Keepers	28
4.6.4 Fees	29
4.7 Swap	29

4.7.1 Order Book Matching	29
4.7.2 Automated Market Makers (AMMs)	30
4.8 Collateralized Loans	32
4.9 Flash Loan (Uncollateralized Loan)	33
5. Problems DeFi Solves	33
5.1 Inefficiency	34
5.1.1 Keepers	34
5.1.2 Forking	34
5.2 Limited Access	35
5.2.1 Yield Farming	35
5.2.1 Initial DeFi Offering	35
5.3 Opacity	36
5.3.1 Smart Contracts	36
5.4 Centralized Control	36
5.4.1 Decentralized Autonomous Organization	37
5.5 Lack of Interoperability	37
5.5.1 Tokenization	37
5.5.2 Networked Liquidity	38
6. DeFi Deep Dive	38
6.1 Credit/Lending	39
6.1.1 MakerDAO	39
6.1.2 Compound	44
6.1.3 Aave	50
6.2 Decentralized Exchange	53
6.2.1 Uniswap	53
6.3 Derivatives	59
6.3.1 Yield Protocol	59
6.3.2 dYdX	62
6.3.3 Synthetix	67
6.4 Tokenization	69
6.4.1 Set Protocol	70
6.4.2 wBTC	71
7. Risks	72
7.1 Smart-Contract Risk	72
7.2 Governance Risk	74

7.3 Oracle Risk	75
7.4 Scaling Risk	76
7.5 DEX Risk	78
7.6 Custodial Risk	74
7.7 Regulatory Risk	80
8. Conclusions: The Losers and the Winners	81

1. Introduction

We have come full circle. The earliest form of market exchange was peer to peer, also known as barter. Barter was highly inefficient because supply and demand had to be exactly matched between peers. To solve the matching problem, money was introduced as a medium of exchange and store of value. Initial types of money were not centralized. Agents accepted any number of items such as stones or shells in exchange for goods. Eventually, specie money emerged, a form in which the currency had tangible value. Today, we have non-collateralized (fiat) currency controlled by central banks. Whereas the form of money has changed over time, the basic infrastructure of financial institutions has not changed.

However, the scaffolding is emerging for a historic disruption of our current financial infrastructure. DeFi or decentralized finance seeks to build and combine open-source financial building blocks into sophisticated products with minimized friction and maximized value to users using blockchain technology. Given it costs no more to provide services to a customer with \$100 or \$100 million in assets, we believe that DeFi will replace all meaningful centralized financial infrastructure in the future. This is a technology of inclusion whereby anyone can pay the flat fee to use and benefit from the innovations of DeFi.

DeFi is fundamentally a competitive marketplace of decentralized financial applications that function as various financial “primitives” such as exchange, save, lend, and tokenize. These applications benefit from the network effects of combining and recombining DeFi products and attracting increasingly more market share from the traditional financial ecosystem.

Our book details the problems that DeFi solves: **centralized control, limited access, inefficiency, lack of interoperability, and opacity**. We then describe the current and rapidly growing DeFi landscape, and present a vision of the future opportunities that DeFi unlocks. Let’s begin with the problems:

Five Key Problems of Centralized Financial Systems

For centuries, we have lived in a world of centralized finance. Central banks control the money supply. Financial trading is largely done via intermediaries. Borrowing and lending is conducted through traditional banking institutions. In the last few years, however, considerable progress has been made on a much different model - decentralized finance or DeFi. In this framework, peers interact with peers via a common ledger that is not controlled by any centralized organization. DeFi offers considerable potential for solving the five key problems associated with centralized finance:

Centralized control. Centralization has many layers. Most consumers and businesses deal with a single, localized bank. The bank controls rates and fees. Switching is possible, but it can be costly. Further, the US banking system is highly concentrated. The four largest banks have a 44%

share of insured deposits compared to 15% in 1984.¹ Interestingly, the US banking system is less concentrated than other countries, such as the United Kingdom and Canada. In a centralized banking system, a single centralized entity attempts to set short-term interest rates and to influence the rate of inflation. The centralization phenomenon does not just pertain to the legacy financial sector. Relatively new tech players dominate certain industries, for example, Amazon (retail) and Facebook/Google (digital advertising).

Limited access. Today, 1.7 billion people are unbanked making it very challenging for them to obtain loans and to operate in the world of internet commerce. Further, many consumers must resort to pay-day lending operations to cover liquidity shortfalls. Being banked, however, does not guarantee access. For example, a bank may not want to bother with the small loan that a new business requires and the bank may suggest a credit card loan. The credit card could have a borrowing rate well above 20% per year, a high hurdle rate for finding profitable investment projects.

Inefficiency. A centralized financial system has many inefficiencies. Perhaps the most egregious example is the credit card interchange rate that causes consumers and small businesses to lose up to 3% of a transaction's value with every swipe due to the payment network oligopoly's pricing power. Remittance fees are 5-7%. Another example is the two days it takes to "settle" a stock transaction (officially transfer ownership). In the internet age, this seems utterly implausible. Other inefficiencies include: costly (and slow) transfer of funds, direct and indirect brokerage fees, lack of security, and the inability to conduct microtransactions. Many of these inefficiencies are not obvious to users. In the current banking system, deposit interest rates remain very low and loan rates high because banks need to cover their bricks-and-mortar costs. A similar issue arises in the insurance industry.

Lack of interoperability. Consumers and businesses deal with financial institutions in an environment that locks interconnectivity. Our financial system is siloed and designed to sustain high switching costs. Moving money from one institution to another can be unduly lengthy and complicated. A wire transfer can take three days to complete. This problem is well-known and some attempts are being made to mitigate it. A recent example is Visa's attempted acquisition of Plaid in 2019. Plaid allows any company to plug into a financial institution's information stack with the user's permission. This is an example of a corporation operating in the world of centralized finance trying to acquire a product to mitigate a particular problem but not addressing the fundamental problems with the current financial infrastructure. It was a strategic move to buy time.

Opacity. The current financial system is not transparent. Bank customers have very little information on the financial health of their bank and must place their faith in the limited government protection of FDIC insurance on their deposits. Bank customers seeking a loan find it difficult to determine if the offered rate is competitive. The market for loans is very fragmented, although the consumer insurance industry has made some progress with fintech services that

¹ See Corbae, Dean and Pablo D'Erasco, 2020, Rising Bank Concentration, Staff Paper #594, Federal Reserve Bank of Minneapolis, March. <https://doi.org/10.21034/sr.594>

offer to find the “lowest” price. The current list of competing lenders, however, all suffer from the system’s inefficiencies. The result is that the “lowest” still reflects legacy bricks-and-mortar costs as well as bloated back-office costs.

The implications of these five problems are twofold. First, many of these costs lead to *lower economic growth*. For example, if loan rates are high because of legacy costs, high-quality investment projects may be foregone, as explained previously. An entrepreneur’s high-quality idea may target a 20% rate of return precisely the type of project that accelerates economic growth. If the bank tells the entrepreneur to borrow money on her credit card at 24% per year, this profitable project may never be pursued.

Second, these problems perpetuate and/or exacerbate *inequality*. Most (across the political spectrum) agree there should be equality of opportunity: a project should be financed based on the quality of the idea and the soundness of the execution plan, and not by other factors. Importantly, inequality also limits growth when good ideas are not financed. While purported to be the “land of opportunity”, the United States has one of the worst records in migrating income from the bottom quartile to the top quartile.² Inequality of opportunity arises, in part, from lack of access to the current banking system, reliance on costly alternative financing such as payday lending and the inability to buy or sell in the modern world of e-commerce.

These implications are far-reaching and, by any calculus, this is a long list of serious problems that are endemic to our current system of centralized finance. While we are in the digital era, our financial infrastructure has failed to fully adopt. Decentralized finance offers new opportunities. The technology is nascent but the upside is promising.

Our book has multiple goals. First, we identify the weaknesses in the current system, including discussion of some early initiatives that challenged the business models of centralized finance. Next, we explore the origins of decentralized finance. We then discuss a critical component of DeFi: blockchain technology. Next, we explore what solutions DeFi offers and couple this with a deep dive on some leading ideas in this emerging space. We then explore the major risk factors. We conclude by looking to the future and attempt to identify the winners and losers.

² See Chetty, R., N. Hendren, P. Kline, and E. Saez (2014), “Where is the land of opportunity? The geography of intergenerational mobility in the United States”, *Quarterly Journal of Economics* 129:4, 1553-1623, and Narayan, A., R. Van der Weide, A. Cojocaru, C. Lakner, S. Redaelli, D. Mahler, R. Ramasubbaiah, and S. Thewissen (2018), *Fair Progress?: Economic Mobility Across Generations Around the World, Equity and Development*, Washington DC: World Bank.

2. The Origins of Modern Decentralized Finance

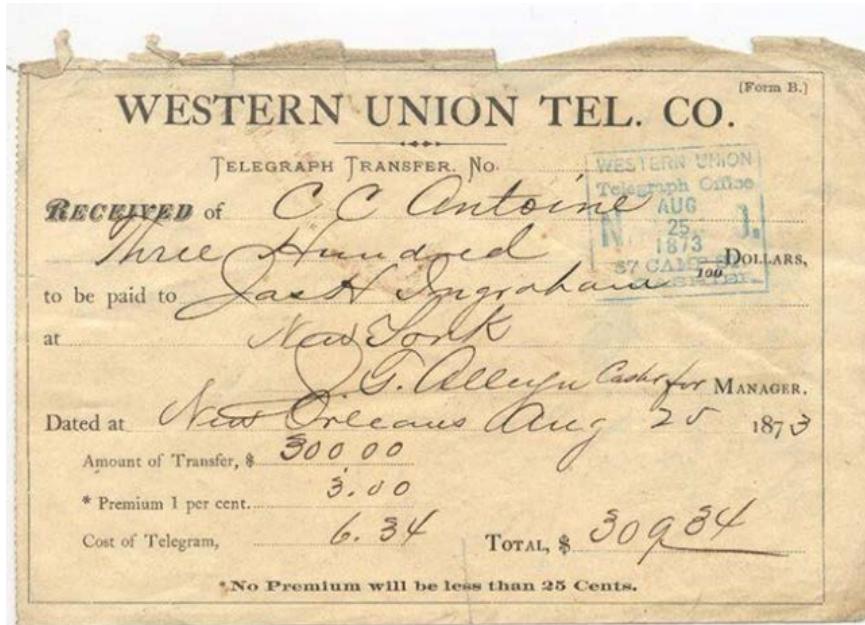
2.1 A brief history of finance

While we argue that today's financial system is plagued with inefficiencies, it is a lot better than systems of the past. As mentioned in the previous chapter, initial market exchanges were peer to peer. A barter system required the exact matching of two parties' needs. Likely at the same time and as response to the inefficiency in the barter system, an informal credit system emerged in villages whereby people kept a mental record of "gifts".³

Coinage came much later with the first modern coins in Lydia around 600 BCE. These coins provided the now traditional functions of money: unit of account, medium of exchange and store of value. Important characteristics of money included: durability, portability, divisibility, uniformity, limited supply, acceptability and stability. Bank notes, originating in China, made their way to Europe in the 13th century.

Non-physical transfer of money originated in 1871 with Western Union. Exhibit 1 shows a copy of an early transfer, for \$300. Notice the fees amount to \$9.34 or roughly 3%. It is remarkable that so little has changed in 150 years. Money transfers are routinely more expensive and credit card fees are 3%.

Exhibit 1: Western Union transfer from 1873



³ See <https://www.creditslips.org/creditslips/2020/06/david-graebers-debt-the-first-5000-years.html>

The pace of innovation increased in the last century: Credit cards (1950) with Diners Card, ATM (1967) by Barclays Bank, telephone banking (1983) from Bank of Scotland, and Internet banking (1994) by Stanford Federal Credit Union. Further innovation, RFID payments (1997) with Mobil Speedpass, chip and pin credit cards (2005), and Apple Pay (2014).

Importantly, all of these innovations were built on the backbone of centralized finance. Indeed, the current system of banking has not changed much in the past 150 years. While digitization was an important innovation, it was an innovation that supported a legacy structure. The high costs associated with the legacy system spurred further innovations that we now refer to as Fintech.

2.2 Fintech

When costs are high, innovation will arise to capitalize on inefficiencies. However, innovation may be slowed by a powerful layer of middle people. An early example of decentralized finance emerged in the foreign currency (forex) market 20 years ago. At the time, large corporations used their investment banks to manage their forex needs. For example, a U.S.-based corporation might need €50 million at the end of September to make a payment on some goods purchased in Germany. Their bank would quote a rate for the transaction. At the same time, another client of the bank might need to sell €50 million at the end of September. The bank would quote a different rate. The difference in the rate is known as the spread and the spread is the profit that the bank makes for being the intermediary. Given the multi-trillion dollar forex market, this was an important part of bank profits.

In early 2001, a fintech startup offered the following idea.⁴ Instead of individual corporations querying various banks to get the best rate, why not have an electronic system match the buyers and sellers directly at an agreed-upon price and *no* spread. Indeed, the bank could offer this service to its own customers and collect a modest fee (compared to the spread). Furthermore, given that some customers deal with multiple banks, it would be possible to connect customers at all banks participating in the peer-to-peer network.

You can imagine the reception. The bank might say: “are you telling me we should invest in an electronic system that will cannibalize our business and largely eliminate a very important profit center?” However, even 20-years ago, banks realized that their largest customers were very unhappy with the current system: as globalization surged these customers faced unnecessary forex transactions costs.

An even earlier example was the rise of dark pool stock trading. In 1979, the US Securities and Exchange Commission instituted Rule 19c3 that allowed stocks listed on one exchange, such as the NYSE, to be traded off-exchange. Many large institutions moved their trading, in particular, large blocks, to these dark pools where they traded peer-to-peer with far lower costs than traditional exchange-based trading.

⁴ See: <https://faculty.fuqua.duke.edu/~charvey/Media/2001/EuromoneyOct01.pdf>

The excessive costs of transacting brought in many fintech innovations. For example, an earlier innovator in the payments space was PayPal, which was founded over 20 years ago.⁵ Even banks have added their own payment systems. For example, in 2017, seven of the largest U.S. banks launched Zelle.⁶ An important commonality of these cost-reducing fintech advances is that these innovations rely on the centralized backbone of the current financial infrastructure.

2.3 Bitcoin and Cryptocurrency

The dozens of digital currency initiatives beginning in the early 1980s all failed.⁷ The landscape shifted, however, with the publication of the famous Satoshi Nakamoto Bitcoin [white paper](#) in 2008. The paper presents a peer-to-peer system that is decentralized and utilizes the concept of blockchain. While blockchain was invented in 1991 by [Haber and Stornetta](#), it was primarily envisioned to be a time-stamping system to keep track of different versions of a document. The key innovation of Bitcoin was to combine the idea of blockchain (time stamping) with a consensus mechanism called *Proof of Work* (introduced by [Back](#) in 2002). The technology produced an immutable ledger that eliminated a key problem with any digital asset - you can make perfect copies and spend them multiple times. Blockchains allow for the key features desirable in a store of value, but which never before were simultaneously present in a single asset. Blockchains allow for cryptographic scarcity (Bitcoin has a fixed supply cap of 21 million), censorship resistance and user sovereignty (no entity other than the user can determine how to use funds), and portability (can send any quantity anywhere for a low flat fee). These features combined in a single technology make cryptocurrency a powerful innovation.

The value proposition of Bitcoin is important to understand, and can be put into perspective by assessing the value proposition of other financial assets. Consider the US dollar, for example. It used to be backed by gold before the gold standard was removed in 1971. Now, the demand for USD comes from: 1) Taxes; 2) Purchase of US goods denominated in USD; and 3) Repayment of debt denominated by USD. None of these three cases create intrinsic value but rather value based on the network that is the US economy. Expansion or contraction in these components of the US economy can impact the price of the USD. Additionally, shocks to the supply of USD adjust its price at a given level of demand. The Fed can adjust the supply of USD through monetary policy to achieve financial or political goals. Inflation eats away at the value of USD, decreasing its ability to store value over time. One might be concerned with runaway inflation, what Paul Tudor Jones calls, “The Great Monetary Inflation”, which would lead to a flight to inflation resistant assets.⁸ Gold has proven to be a successful inflation hedge due to its practically limited supply, concrete

⁵ PayPal founded as Confinity in 1998 did not begin offering a payments function until it merged with X.com in 2000.

⁶ Other examples include: Cash App, Braintree, Venmo, and Robinhood.

⁷ See Harvey, C. R., The history of digital money (2020),

https://faculty.fuqua.duke.edu/~charvey/Teaching/697_2020/Public_Presentations/697/History_of_Digital_Money_2020_697.pdf

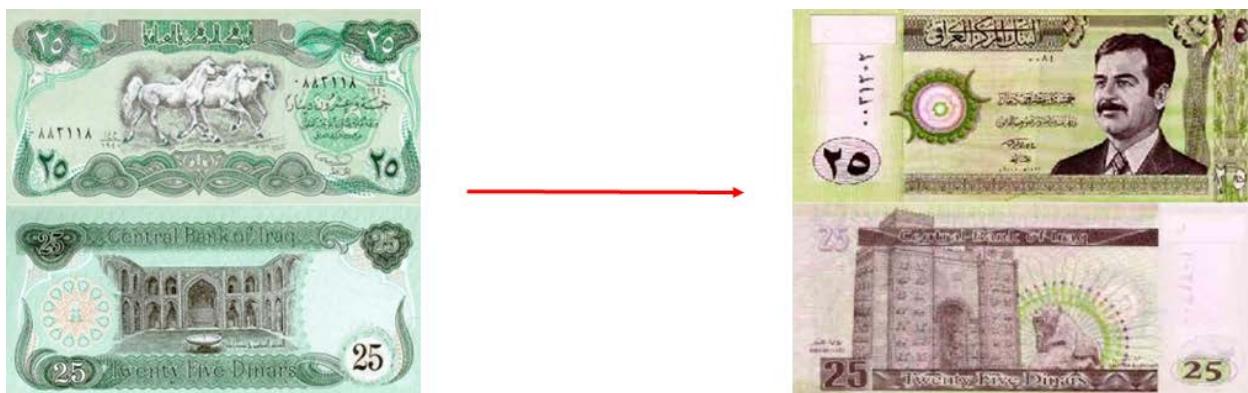
⁸ <https://www.lopp.net/pdf/BVI-Macro-Outlook.pdf>

utility, and general global trustworthiness. However, given that gold is a volatile asset, its historical hedging ability is only realized at extremely long horizons.⁹

Many argue that bitcoin has no “tangible” value and therefore it should be worthless. Continuing the gold analogy, approximately two thirds of gold is used for jewelry and some is used in technology hardware. Gold has tangible value. The US dollar, while a fiat currency, has value as “legal tender”. However, there are many examples from history whereby currency emerged without any backing that had value.

A relatively recent example is the Iraqi Swiss dinar. This was the currency of Iraq until the first Gulf War in 1990. The printing plates were manufactured in Switzerland (hence the name) and the printing was outsourced to the U.K. In 1991, Iraq was divided with the Kurds controlling the north and Saddam Hussien in the south. Due to sanctions, Iraq could not import dinars and had to start local production. In May 1993, the Central Bank of Iraq announced that citizens had three weeks to exchange old 25 dinars for new ones (Exhibit 2).

Exhibit 2: Iraqi Swiss dinars and new dinars



The old Swiss dinar continued to be used in the north. In the south, the new dinar suffered from extreme inflation. Eventually, the exchange rate was 300 new dinars for a single Iraqi Swiss dinar. The key insight here is that the Iraqi Swiss dinar had no official backing - but it was accepted as money. There was no tangible value yet it had fundamental value. Importantly, value can be derived from both tangible and intangible value.

The features of Bitcoin that we have mentioned, particularly scarcity and self-sovereignty, make it a potential store of value and possible hedge to political and economic unrest at the hands of global governments. As the network grows, the value proposition only increases due to increased trust and liquidity. Although Bitcoin was originally intended as a peer-to-peer currency, its deflationary characteristics and flat fees discourage its use in small transactions. We argue that Bitcoin is the flagship of a new asset class, namely cryptocurrencies, which can have varied use

⁹ C. Erb and Harvey, C. R., (2013) The Golden Dilemma, *Financial Analysts Journal*, 69:4, pp. 10-42, show that gold is an unreliable inflation hedge over short and medium term horizons.

cases based on the construction of their networks. Bitcoin itself, we believe will continue to grow as an important store of value and a potential inflation hedge - over long horizons.¹⁰

The original cryptocurrencies offered an alternative to a financial system that had been dominated by governments and centralized institutions such as central banks. They arose largely from a desire to replace inefficient, siloed financial systems with immutable, borderless, open-source algorithms. The currencies can adjust their parameters such as inflation and mechanism for consensus via their underlying blockchain to create different value propositions. We will discuss blockchain and cryptocurrency in greater depth in section 3 and, for now, we will focus on a particular cryptocurrency with special relevance to DeFi.

2.4 Ethereum and DeFi

Ethereum (ETH) is currently the second largest cryptocurrency by market cap (\$230b). Vitalik Buterin introduced the idea in 2014 and Ethereum mined its first block in 2015. Ethereum is in some sense a logical extension of the applications of Bitcoin. It allows for *smart contracts* - which are code that lives on a blockchain, can control assets and data, and define interactions between the assets, data, and network participants. The capacity for smart contracts defines Ethereum as a *smart contract platform*.

Ethereum and other smart contract platforms specifically gave rise to the *decentralized application* or *dApp*. The backend components of these applications are built with interoperable, transparent smart contracts that continue to exist as long as the chain they live on exists. dApps allow peers to interact directly and remove the need for a company to act as a central clearing house for app interactions. It quickly became apparent that the first killer dApps would be financial ones.

The drive toward financial dApps became a movement in its own right known as *decentralized finance* or *DeFi*. DeFi seeks to build and combine open-source financial building blocks into sophisticated products with minimized friction and maximized value to users. Because it costs no more at an organization level to provide services to a customer with \$100 or \$100 million in assets, DeFi proponents believe that all meaningful financial infrastructure will be replaced by smart contracts which can provide more value to a larger group of users. Anyone can simply pay the flat fee to use the contract and benefit from the innovations of DeFi. We will discuss smart contract platforms and dApps in more depth in chapter 3.

DeFi is fundamentally a competitive marketplace of financial dApps that function as various financial “primitives” such as exchange, lend, tokenize, and so forth. These dApps benefit from the network effects of combining and recombining DeFi products and attracting increasingly more market share from the traditional financial ecosystem. Our goal is to give an overview of the

¹⁰ Similar to gold, bitcoin is likely too volatile to be a reliable inflation hedge over short horizons. While theoretically decoupled from any country's money supply or economy, in the brief history of bitcoin, we have not experienced any inflation surge. So there is no empirical evidence of its efficacy.

problems that DeFi solves, describe the current and rapidly growing DeFi landscape, and present a vision of the future opportunities that DeFi unlocks.

3. DeFi Infrastructure

In this chapter, we discuss the innovations that led to DeFi and lay out the terminology.

3.1 Blockchain

The key to all DeFi is the decentralizing backbone, a blockchain. Blockchains are fundamentally software protocols that allow multiple parties to operate under shared assumptions and data without trusting each other. These data can be anything, such as location and destination information of items in a supply chain or account balances of a token. Updates are packaged into “blocks” and are “chained” together cryptographically to allow an audit of the prior history, hence the name.

The reason blockchains are possible is a *Consensus Protocol*, a set of rules that determine what kinds of blocks can become part of the chain and become the “truth”. These consensus protocols are designed to be resistant to malicious tampering up to a certain security bound. The blockchains we focus on use the *Proof of Work (PoW)* consensus protocol, which relies on a computationally-intensive lottery to determine which block to add. The participants agree that the *longest chain* of blocks is the truth. If an attacker wants to make a longer chain that contains malicious transactions, they have to outpace all of the computational *work* of the entire rest of the network. In theory, they would need a majority of the network power to accomplish this, hence the famous *51% attack* being the boundary of PoW security. Luckily, it is extraordinarily difficult for any actor, even an entire country, to amass this much network power on the most widely-used blockchains such as Bitcoin or Ethereum. Even if a majority of the network power (“hashrate”), can be temporarily acquired, the extent of block history that can be overwritten is constrained by how long this majority can be maintained.

While we focus on Proof of Work, there are many alternative consensus mechanisms with the most important being *Proof of Stake (PoS)*. In Proof of Stake, validators commit some capital (the stake) to attest that the block is valid. Validators make themselves available by staking their cryptocurrency and then they may be selected to propose a block. The proposed block needs to be attested by a majority of the other validators. Validators profit by both proposing a block as well as attesting to the validity of others’ proposed blocks.

As long as no malicious party can acquire majority control of the network computational power, then transactions will be processed by the good-faith actors and appended to the ledger when a block is “won”.

3.2 Cryptocurrency

The most popular application of blockchain technology is cryptocurrency. Cryptocurrency is a token (usually scarce) that is cryptographically secured and transferred. The scarcity is what assures the possibility of value, and is itself an innovation of blockchain. Typically digital objects are easily copied. As Eric Schmidt, the former CEO of Google has said,¹¹ “[Bitcoin] is a remarkable cryptographic achievement and the ability to create something that is not duplicable in the digital world has enormous value.”

No one can post a false transaction without ownership of the corresponding account due to the *asymmetric key cryptography* protecting the accounts. You have one “public” key representing an address to receive tokens, and a “private” key used to unlock and spend tokens you have custody over. This same type of cryptography is used to protect your credit card information and data when using the internet. A single account cannot “double-spend” their tokens because the ledger keeps an audit of their balance at any given time and the faulty transaction would not clear. The ability to prevent “double-spend” without a central authority illustrates the primary advantage of using a blockchain to maintain the underlying ledger.

The initial cryptocurrency model is the Bitcoin blockchain, which functions almost exclusively as a payment network, with the capabilities of storing and transacting bitcoins across the globe in real-time with no intermediaries or censorship. This is the powerful value proposition that gives bitcoin its value. Even though its network effects are strong, technological competitors do offer enhanced functionality.

3.3 The Smart Contract Platform

A crucial ingredient of DeFi is a *smart contract* platform. These blockchains go beyond a simple payments network such Bitcoin and allow for the creation of smart contracts that enhance the capabilities of the chain itself. Ethereum is the primary example of a smart contract platform. A smart contract is code that can create and transform arbitrary data or tokens on top of the blockchain of which it is a part. The concept is powerful because it allows the user to trustlessly encode rules for any type of transaction and even create scarce assets with specialized functionality. Many of the clauses of traditional business agreements could be shifted to a smart contract, which not only would enumerate, but algorithmically enforce those clauses. Smart contracts go beyond finance, and have applications in gaming, data stewardship and supply chain among other purposes.

An interesting caveat applies to Ethereum, but not necessarily to all smart contract platforms, is the existence of a transaction fee known as a *gas fee*. Imagine Ethereum as one giant computer with many applications (smart contracts). If someone wants to use the computer, they must pay a fee for each unit of computation they use. A simple computation, such as sending ETH, requires

¹¹ From a panel discussion at the Computer History Museum in 2014. See:
<https://www.newsbtc.com/news/google-chairman-eric-schmidt-bitcoin-architecture-amazing-advancement/>

minimal work updating a few account balances. This has a relatively small gas fee. A complex computation that involves minting tokens and checking various conditions across many contracts costs correspondingly more gas.

A helpful analogy for Ethereum is a car. If someone wants to drive a car, a certain amount of gas is needed and there is a fee to acquire the gas. The gas fee may lead to a poor user experience, however. The gas fee forces agents to maintain an ETH balance in order to pay it and to worry not only about overpaying but also underpaying and having the transaction not take place at all. For this reason, initiatives are ongoing to abstract away gas fees from end users and support competitor chains that completely remove this concept of gas. Gas is important, however, as a primary mechanism for preventing attacks on the system that generate an *infinite loop* of code. It is not feasible to identify malicious code of this kind before running it, a problem formally known in computer science as *the halting problem*. Gas secures the Ethereum blockchain by making such attacks prohibitively expensive. Continuing our analogy, gas solves the halting problem in the following way: Suppose a “car” is on autopilot stuck in full throttle with no driver. Gas acts as a limiting factor, because the car has to stop eventually when the gas tank empties. This incentivizes highly efficient smart contract code, as contracts that utilize fewer resources and reduce the probability of user failures have a much higher chance of being used and succeeding in the market.

On a smart contract platform, the possibilities rapidly expand beyond what developers desiring to integrate various applications can easily handle. This leads to the adoption of standard interfaces for different types of functionality. On Ethereum these standards are called *Ethereum Request for Comments (ERC)*. The best known of these define different types of tokens that have similar behavior. ERC-20 is the standard for fungible tokens,¹² it defines an interface for tokens whose units are identical in utility and functionality. It includes behavior such as transferring units and approving operators for using a certain portion of a user’s balance. Another is ERC-721, the non-fungible token standard. ERC-721 tokens are unique, and are often used for collectibles or assets such as P2P loans. The benefit of these standards is that application developers can code for one interface, and support every possible token that implements that interface. We will discuss these interfaces further in Section 4.

3.4 Oracles

An interesting problem with blockchain protocols is that they are isolated from the world outside of their ledger. That is, the Ethereum blockchain only authoritatively knows what is happening on the Ethereum blockchain, and not, for example, the level of the S&P 500 or which team won the Super Bowl. This limitation constrains applications to Ethereum native contracts and tokens thus reducing the utility of the smart contract platform and is generally known as the *oracle problem*. An *oracle*, in the context of smart contract platforms, is any data source for reporting information

¹² Fungible tokens have equal value just as every dollar bill has equal value and a \$10 dollar bill is equal to two \$5 dollar bills. Non-fungible tokens, in contrast, reflect the value of what they are associated with (e.g., one non-fungible token may be associated with a piece of art like a painting). They do not necessarily have equal value.

external to the blockchain. How can we create an oracle that can authoritatively speak about off-chain information in a trust-minimized way? Many applications require an oracle, and the implementations exhibit varying degrees of centralization.

There are several implementations of oracles in various DeFi applications. A common approach is for an application to host its own oracle or hook into an existing oracle from a well-trusted platform. One Ethereum-based platform known as [Chainlink](#) is designed to solve the oracle problem by using an aggregation of data sources. The Chainlink whitepaper includes a reputation-based system, which has not yet been implemented. We discuss the oracle problem later in more depth. Oracles are surely an open design question and challenge for DeFi to achieve utility beyond its own isolated chain.

3.5 Stablecoins

A crucial shortcoming to many cryptocurrencies is excessive volatility. This adds friction to users who wish to take advantage of DeFi applications but don't have the risk-tolerance for a volatile asset like ETH. To solve this, an entire class of cryptocurrencies called stablecoins has emerged. Stablecoins are intended to maintain price parity with some target asset, USD or gold for instance. Stablecoins provide the necessary stability that investors seek to participate in many DeFi applications and allow a cryptocurrency native solution to exit positions in more volatile cryptoassets. They can even be used to provide on-chain exposure to the returns of an off-chain asset if the target asset is not native to the underlying blockchain (e.g., gold, stocks, ETFs). The mechanism by which the stablecoin maintains its peg varies by implementation. The three primary mechanisms are fiat-collateralized, crypto-collateralized, and non-collateralized stablecoins.

By far the largest class of stablecoins are fiat-collateralized. These are backed by an off-chain reserve of the target asset. Usually these are custodied by an external entity or group of entities which undergo routine audits to verify the collateral's existence. The largest fiat-collateralized stablecoin is [Tether](#) (USDT) with a market capitalization of \$24 billion dollars, making it the third largest cryptocurrency behind Bitcoin and Ethereum at time of writing. Tether also has the highest trading volume of any cryptocurrency but is not audited. The second-largest is [USDC](#), backed by Coinbase and Circle is audited. USDC is redeemable 1:1 for USD and vice-versa for no fee on Coinbase's exchange. USDT and USDC are very popular to integrate into DeFi protocols as demand for stablecoin investment opportunities is high. There is an inherent risk to these tokens however as they are centrally controlled and maintain the right to blacklist accounts.

The second largest class of stablecoins are crypto-collateralized. These are stablecoins which are backed by an overcollateralized amount of another cryptocurrency. Their value can be hard or soft pegged to the underlying asset depending on the mechanism. The most popular crypto-collateralized stablecoin is DAI, created by [MakerDAO](#) and it is backed by mostly ETH with collateral support for a few other cryptoassets. It is soft pegged with economic mechanisms that incentivize supply and demand to drive the price to \$1. DAI's market capitalization is \$1 billion as of writing. We will do a deep dive into MakerDAO and DAI in section 6.1. Another popular crypto-

collateralized stablecoin is sUSD. This is part of the [Synthetix](#) platform we explore in section 6.6. It is backed by the Synthetix network token (SNX) and is hard-pegged to 1 USD through their exchange functionality. Crypto-collateralized stablecoins have the advantages of decentralization and secured collateral. The drawback is that their scalability is limited. To mint more of the stablecoin, a user must necessarily back the issuance by an overcollateralized debt position. In some cases like DAI there is even a debt ceiling that further limits the supply growth.

The last and perhaps the most interesting class of stablecoins are non-collateralized. These are not backed by any underlying asset, and use algorithmic expansion and contraction of supply to shift the price to the peg. They often employ a seigniorage model where the token holders in the platform receive the increase in supply when demand increases. When demand decreases and the price slips below the peg, these platforms would issue bonds of some form which entitle the holder to future expansionary supply before the token holders receive their share. This mechanism works almost identically to the central bank associated with fiat currencies, with the caveat that these platforms have an explicit goal of pegging the price rather than funding government spending or other economic goals. A noteworthy early example of an algorithmic stablecoin is [Basis](#), which had to close down due to regulatory hurdles. Current examples of algorithmic stablecoins include [Ampleforth](#) (AMPL) and [Empty Set Dollar](#) (ESD). The drawback to non-collateralized stablecoins is that they have a lack of inherent underlying value backing the exchange of their token. In contractions, this can lead to “bank runs” in which the majority of holders are left with large sums of the token which are no longer worth the peg price.

It is still an open problem to create a decentralized stablecoin which both scales efficiently and is resistant to collapse in contractions. Further, there are regulatory issues which we will discuss later.¹³ Stablecoins are an important component of DeFi infrastructure as they allow users to benefit from the functionality of the applications without risking unnecessary price volatility.

3.6 Decentralized Applications

As mentioned earlier, dApps are a critical DeFi ingredient. dApps are similar to traditional software applications except they live on a decentralized smart contract platform. The primary benefit of these applications is their *permissionlessness* and *censorship-resistance*. Anyone can use them, and no single body controls them. A separate but related concept is a *decentralized autonomous organization (DAO)*. A DAO has its rules of operation encoded in smart contracts that determine who can execute what behavior or upgrade. It is common for a DAO to have some kind of *governance token*, which gives an owner some percentage of the vote on future outcomes. We will explore governance in much more detail later.

¹³ See, e.g., Financial Stability Board, [“Regulation, Supervision and Oversight of ‘Global Stablecoin’ Arrangements”](#), October 2020.

4. DeFi Primitives

Now that we have laid the groundwork by detailing the DeFi infrastructure, in this chapter we will describe the primitive financial actions that developers can use. A developer can combine these actions to create complex dApps. We will explain in detail each of the primitive actions and the advantages each may have over its centralized counterparts.

4.1 Transactions

Ethereum transactions are the atoms of DeFi (and Ethereum as a whole). Transactions involve sending data and/or ETH (or other tokens) from one address to another. All Ethereum interactions, including each of the primitives discussed in this section, begin with a transaction. Therefore, good comprehension of the mechanics of transactions is crucial to understanding Ethereum, in particular, and DeFi, in general.

An Ethereum user can control addresses through an *externally owned account* (EOA) or by using smart contract code (*contract account*). When data is sent to a contract account, the data are used to execute code in that contract. The transaction may or may not have an accompanying ETH payment for use by the contract. Transactions sent to an EOA can only transfer ETH.¹⁴

A single transaction starts with an end-user from an EOA, but can interact with a large number of dApps (or any Ethereum smart contract) before completing. The transaction starts by interacting with a single contract, which will enumerate all of the intermediate steps in the transaction required within the contract body.

Clauses in a smart contract can cause a transaction to fail and thereby revert all previous steps of the transaction; as a result, transactions are *atomic*. Atomicity is a critical feature of transactions because funds can move between many contracts (i.e., “exchange hands”) with the knowledge and security that if one of the conditions is not met, the contract terms reset as if the money never left the starting point.

As we mentioned in Section 3, transactions have a gas fee, which varies based on the complexity of the transaction. When, for example, ETH is used to compensate a miner for including and executing a transaction, the gas fee is relatively low. Longer or more data-intensive transactions cost more gas. If a transaction reverts for any reason, or runs out of gas, the miner forfeits all gas used until that point. Forfeiture protects the miners who, without this provision, could fall prey to large volumes of failed transactions for which they would not receive payment.

¹⁴ Technically, a transaction sent to an EOA can also send data, but the data have no Ethereum-specific functionality.

The gas price is determined by the market and effectively creates an auction for inclusion in the next Ethereum block. Higher gas fees signal higher demand and therefore generally receive higher priority for inclusion.

A technical aside about transactions is that they are posted to a *memory pool*, or *mempool*, before they are added to a block. Miners monitor these posted transactions, add them to their own mempool, and share the transaction with other miners to be included in the next available block. If the gas price offered by the transaction is uncompetitive relative to other transactions in the mempool, the transaction is deferred to a future block.

Any actor can see transactions in the mempool by running or communicating with mining nodes. This visibility can even allow for advanced front-running and other competitive techniques that aid the miner in profiting from trading activity. If a miner sees a transaction in the mempool she could profit from by either executing herself or front-running it, the miner is incentivized to do so if lucky enough to win the block. Any occurrence of direct execution is known as *miner extractable value* (MEV). MEV is a drawback to the proof-of-work model. Certain strategies, such as obfuscating transactions, can mitigate MEV, thus hiding from miners how they might profit from the transactions.

4.2 Fungible Tokens

Fungible tokens are a cornerstone of the value proposition of Ethereum and DeFi. Any Ethereum developer can create a token divisible to a certain decimal granularity and with units that are all identical and interchangeable. By way of example, USD is a fungible asset because one \$100 bill is equivalent to one hundred \$1 bills. As we mentioned in Section 3, the Ethereum blockchain token interface is [ERC-20](#). An interface from an application developer's perspective is the minimum required set of functionality. When a token implements the ERC-20 interface, any application that generically handles the defined functionality can instantly and seamlessly integrate with the token. Using ERC-20 and similar interfaces, application developers can confidently support tokens that do not yet exist.

The ERC-20 interface defines the following core functionality:

- `totalSupply()`—read the token's total supply;
- `balanceOf(account)`—read the balance of the token for a particular account;
- `transfer(recipient address, amount)`—send “amount” tokens from the transaction sender to “recipient address”;
- `transferFrom(sender address, recipient address, amount)`—send “amount” tokens from the balance of tokens held at “sender address” to “recipient address”;
- `approve(spender, amount)`—allows “spender” to spend “amount” tokens on behalf of the account owner;
- `allowance(owner address, spender address)`—returns the amount of tokens the “spender address” can spend on behalf of the “owner address”.

The contract will reject transfers involving insufficient balances or unauthorized spending. The first four functions are intuitive and expected (reading supply, balances, and sending tokens). The last two functions, approve and allowance, are critical to understanding the power of the ERC-20 interface. Without this function, users would be limited to directly transferring tokens to and from accounts. With approval functionality, contracts (or trusted accounts) can be whitelisted to act as custodians for a user's tokens without directly holding the token balance. This functionality widens the scope of possible applications because users retain full custody before an approved spender executes a transaction.

We will now define three main categories of ERC-20 tokens. An ERC-20 token can simultaneously be in more than one category.

4.2.1 Equity Token

An equity token, not to be confused with equities or stocks in the traditional finance sense, is simply a token that represents ownership of an underlying asset or pool of assets. The units must be fungible so that each corresponds to an identical share in the pool. For example, suppose a token, TKN, has a total fixed supply of 10,000, and TKN corresponds to an ETH pool of 100 ETH held in a smart contract. The smart contract stipulates that for every unit of TKN it receives, it will return a pro rata amount of ETH, fixing the exchange ratio at 100 TKN/1 ETH.

We can extend the example so the pool has a variable amount of ETH. Suppose the ETH in the pool increases at 5% per year by some other mechanism. Now 100 TKN would represent 1 ETH plus a 5% perpetuity cash flow of ETH. The market can use this information to accurately price the value of TKN.

In actual equity tokens, the pools of assets can contain much more complex mechanics, going beyond a static pool or fixed rates of increase. The possibilities are limited only by what can be encoded into a smart contract. We will examine a contract with variable interest-rate mechanics in Section 6.1.2, when discussing Compound, and a contract that owns a multi-asset pool with a complex fee structure in Section 6.2.1 when discussing Uniswap. In Section 6.4.1 we explain Set Protocol, which defines a standard interface for creating equity tokens with static or dynamic holdings.

4.2.2 Utility Tokens

Utility tokens are in many ways a catchall bucket, although they do have a clear definition. Utility tokens are fungible tokens that are required to utilize some functionality of a smart contract system or that have an intrinsic value proposition defined by its respective smart contract system. In many cases, utility tokens drive the economics of a system, creating scarcity or incentives where intended by the developers. In some cases, ETH could be used in place of a utility token, but utility tokens allow systems to accrue and maintain decoupled economic value from Ethereum as a whole. A use case that requires a distinct utility token would include a system with algorithmically varied supply. We will discuss the mechanics in more depth in Section 4.5.

The following are examples of use cases for utility tokens:

- To be collateral (e.g., SNX)
- To represent reputation or stake (e.g., REP, LINK)
- To maintain stable value relative to underlying or peg (e.g., DAI, Synthetix Synth)
- To pay application-specific fees (e.g., ZRX, DAI, LINK)

The last example includes all stablecoins, regardless of whether the stablecoin is fiat collateralized, crypto-collateralized, or algorithmic. In the case of USDC, a fiat-collateralized stablecoin, the utility token operates as its own system without any additional smart-contract infrastructure to support its value. The value of USDC arises from the promise of redemption for USD by its backing companies, including Coinbase.

Far more possibilities exist for utility tokens than the few we have mentioned here. Innovation will expand this category as novel economic and technical mechanisms emerge.

4.2.3 Governance Tokens

Governance tokens are similar to equity tokens in the sense they represent percentage ownership. Instead of asset ownership, governance token ownership applies to voting rights, as the name suggests. We start by motivating the types of changes on which owners can vote.

Many smart contracts have embedded clauses stipulating how the system can change; for instance, allowed changes could include adjusting parameters, adding new components, or even altering the functionality of existing components. The ability of the system to change is a powerful proposition given the possibility that the contract a user interacts with today could change tomorrow. In some cases, only developer admins, who encode special privileges for themselves, can control changes to the platform.

Any platform with admin-controlled functionality is not truly DeFi because of the admins' centralized control. A contract without the capacity for change is necessarily rigid, however, and has no way to adapt to bugs in the code or changing economic or technical conditions. For this reason, many platforms strive for a decentralized upgrade process, often mediated by a governance token.

The owners of a governance token would have pro-rata voting rights for implementing any change allowed by the smart contracts that govern the platform. We will discuss the voting mechanisms in Section 5 when we discuss *decentralized autonomous organizations* (DAOs).

A governance token can be implemented in many ways—with a static supply, an inflationary supply, or even a deflationary supply. A static supply is straightforward: purchased shares would correspond directly to a certain percentage control of the vote. The current implementation of the MKR token for MakerDAO has a generally static supply. In Section 6.1 we will take a deep dive on MakerDAO and discuss its implementation in more detail.

Many platforms issue the governance token via an inflation schedule that incentivizes users to utilize particular features of the platform, ensuring the governance token is distributed directly to users. Compound, for example, uses an inflationary implementation approach with its COMP token, which we will discuss in Section 6.2. A deflationary approach would likely consist of using the governance token also as a utility token to pay fees to the platform. These fees would be burned, or removed, from the supply rather than going to a specific entity. The MKR token of MakerDAO used to be burned in this manner in an older version of the platform. We will discuss burning mechanics further in Section 4.5.

4.3 Nonfungible Tokens

As the name suggests, a nonfungible token's units are not equal to the units of other tokens.

4.3.1 NFT Standard

On Ethereum, the [ERC-721](#) standard defines nonfungibility. This standard is similar to ERC-20, except each unit has its own unique ID, rather than all units being stored as a single balance. This unique ID can be linked to additional metadata that differentiate the token from others' stemming from the same contract. Under the `balanceOf(address)` method, the total number of nonfungible tokens (NFTs) in the given contract that the address owns is returned. An additional method, `ownerOf(id)`, returns a specific token, referenced by its ID, that the address owns. Another important difference is that ERC-20 allows for the partial approval of an operator's token balances, whereas ERC-721 uses an all-or-nothing approach. An operator approved to use the NFTs can move any of them.

NFTs have interesting applications in DeFi. Their alternate name, *deeds*, implies their use case as representing unique ownership of unitary assets; an example could be ownership of a particular P2P loan with its own rates and terms. The asset could then be transferred and sold via the ERC-721 interface. Another use case might be to represent a share in a lottery. Lottery tickets could be considered nonfungible because only one or a limited number will be winning tickets and the remainder are worthless. NFTs also have a strong use case in their ability to bridge financial and nonfinancial use cases via *collectibles* (e.g., a token could represent ownership in a piece of art). NFTs can also represent scarce items in a game or other network and retain economic value in secondary markets for NFTs.

4.3.2 Multi-Token Standard

ERC-20 and ERC-721 tokens require an individual contract and address deployed to the blockchain. These requirements can be cumbersome for systems that have many tokens, which are closely related, possibly even a mix of fungible and nonfungible token types. The [ERC-1155](#) token standard resolves this complexity by defining a multi-token model in which the contract holds balances for a variable number of tokens, which can be fungible or nonfungible. The standard also allows for batch reading and transfers, which saves on gas costs and leads to a

smoother user experience. Under ERC-1155 and similar to ERC-721, operators are approved for all supported tokens in a binary all-or-none fashion.

4.4 Custody

A critical DeFi primitive is the ability to escrow or custody funds directly in a smart contract. This is distinct from the situation in ERC-20 when operators are approved to transfer a user's balance. In that case, the user still retains custody of his funds and could transfer the balance at any time or revoke the contract's approval. When a smart contract has full custody over funds, new capabilities (and additional primitives) are possible:

- Retaining fees and disbursing incentives (Section 4.6)
- Facilitation of token swaps (Section 4.7)
- Market making of a bonding curve (Section 4.5)
- Collateralized Loans (Section 4.8)
- Auctions
- Insurance funds

In order to effectively custody tokens, a contract must be programmed to handle the token interface of the corresponding type of token, which would be ERC-20 for fungible tokens and ERC-721 for nonfungible tokens. The contract could generically handle all tokens of that interface or of a specific subset only. Users must exercise caution when sending tokens to contracts because the tokens could become permanently custodied if the contract has no encoded mechanism for releasing the funds of that particular token. Safety checks are often embedded in the token transfer to verify if the contract is registered to support a given token interface as a means to mitigate this potential problem.

4.5 Supply Adjustment

Supply adjustment applies specifically to fungible tokens and the ability to create (*mint*) and reduce (*burn*) supply via a smart contract. We will explore the basic primitives of burning and minting, and a more complex system known as a *bonding curve*.

4.5.1 Burn - Reduce Supply

To burn a token means to remove it from circulation. Burning a token can take two forms. One method is to manually send a token to an unowned Ethereum address. Another, and even more efficient, method is to create a contract that is incapable of spending them. Either approach renders the burned tokens unusable, although the decrease in circulating supply would not be "known" by the token contract. Burning is analogous to the destruction or irreversible loss of currency in the traditional finance world, which is unknown to the issuing government. In practice, ETH or ERC-20 tokens have frequently and accidentally been burned using both forms.

Mechanisms are in place to protect against accidental burning, such as checksumming addresses¹⁵ and registering contracts¹⁶ as being able to handle certain tokens.

More common and useful is the ability to intentionally burn tokens as a part of the smart contract design. Here are some example use cases for burning tokens algorithmically:

- Represent exiting of a pool and redemption of underlying (common in equity tokens like cTokens for Compound discussed in Section 6.1.2)
- Increase scarcity to drive the price upward (e.g., AAVE in Section 6.1.3, Seigniorage Stablecoin models like Basis/ESD)
- Penalize bad acting (discussed further in Section 4.7)

4.5.2 Mint - Increase Supply

The flip side of burning is *minting* tokens. Minting increases the number of tokens in circulation. Contrary to burning, there is no mechanism for accidentally or manually minting tokens. Any mint mechanics have to be directly encoded into the smart contract mechanism. There are many use cases for minting as it can incentivize a wider range of user behavior. Here are some examples:

- Represent entering a pool and acquiring corresponding ownership share (common in equity tokens like cTokens for Compound)
- Decrease scarcity (increase supply) to drive the price downward (seigniorage Stablecoin models like Basis/ESD)
- Reward user behavior

Rewarding user behavior with increases in supply (*inflationary rewards*) has become a common practice to encourage actions such as supplying liquidity or using a particular platform. Consequently, many users engage in *yield farming*, taking actions to seek the highest possible rewards. Platforms can bootstrap their networks by issuing a token with an additional value proposition in the network. Users can keep the token and use it in the context of the network or sell it for a profit. Either way, utilization of the token benefits the platform by increasing activity.

4.5.3. Bonding Curve - Pricing Supply

One advantage of being able to adjust supply up and down on a contractual basis is being able to define a bonding curve. A bonding curve is the price relationship between the token supply and a corresponding asset used to purchase the token(s). In most implementations investors sell back

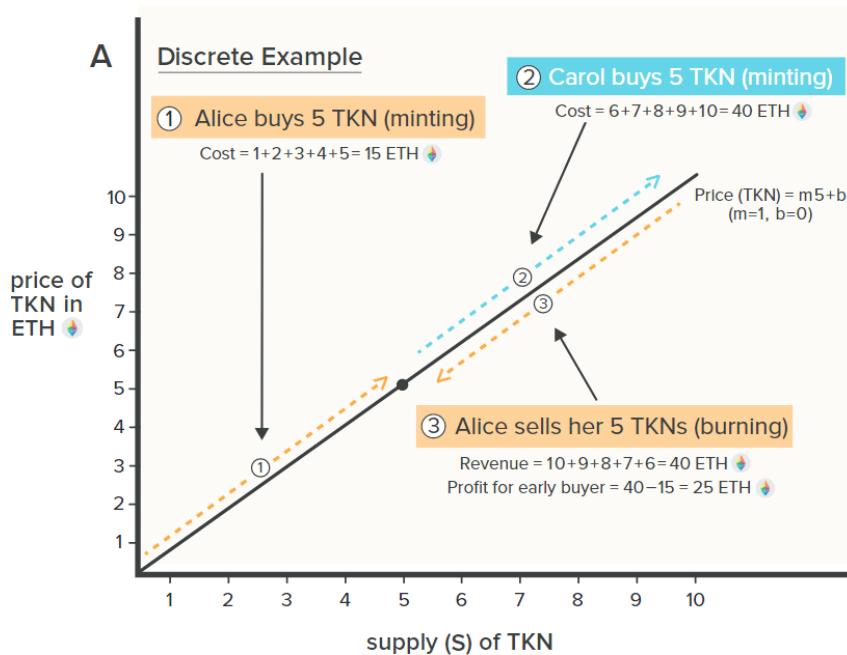
¹⁵ Checksums in general are cryptographic primitives used to verify data integrity. In the context of Ethereum addresses, [EIP-55](#) proposed a specific checksum encoding of addresses to stop incorrect addresses' receiving token transfers. If an address used for a token transfer does not include the correct checksum metadata, the contract assumes the address was mistyped and the transaction would fail. Typically, these checks are added by code compilers before deploying smart contract code and by client software used for interacting with Ethereum.

¹⁶ Registry contracts and interfaces allow a smart contract on chain to determine if another contract it interacts with is implementing the intended interface. For example, a contract may register itself as being able to handle specific ERC-20 tokens if unable to handle all ERC-20 tokens. Sending contracts can verify that the recipient does support ERC-20 tokens as a precondition for clearing the transfer. [EIP-165](#) proposes a standard solution in which each contract declares which interfaces they implement.

to the curve using the same price relationship. The relationship is defined as a mathematical function or as an algorithm with several clauses.

To illustrate, we can use TKN to denote the price of a token denominated in ETH (which could be any fungible cryptoasset) and use S to represent the supply. The simplest possible bonding curve would be $TKN = 1$ (or any constant). This relationship—TKN backed by a constant ratio of ETH—enforces that TKN is pegged to the price of ETH. The next-level bonding curve could be a simple linear bonding curve, where m and b represent the slope and intercept, respectively, in a standard linear function. If $m = 1$ and $b = 0$, the first TKN would cost 1 ETH, the second would cost 2 ETH, and so on. A monotonically increasing bonding curve rewards early investors, because any incremental demand beyond their purchase price would allow them to sell back against the curve at a higher price point.

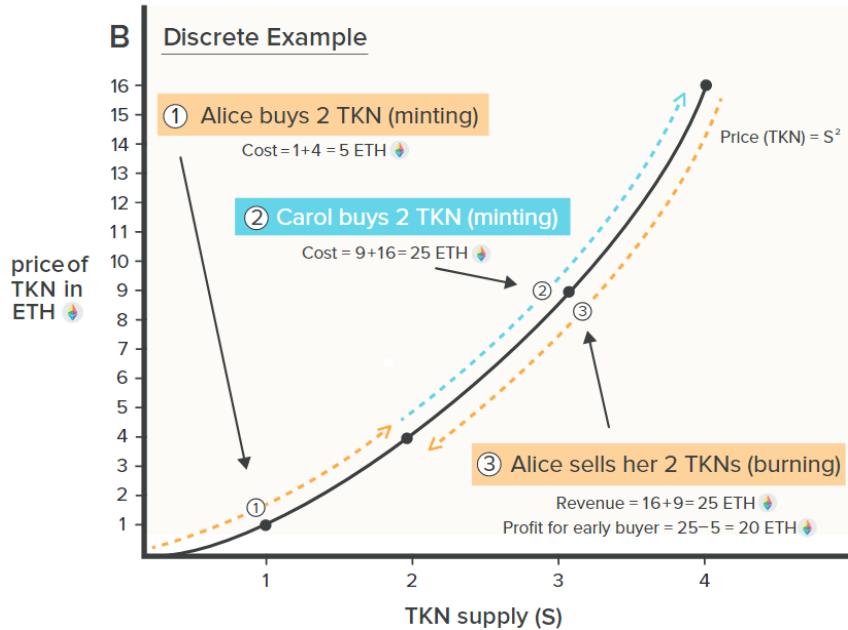
Linear Bonding Curve



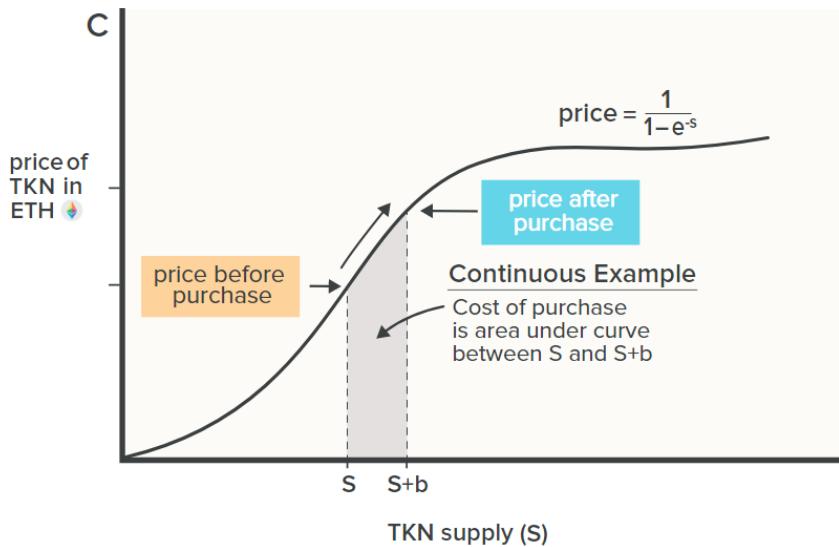
The mechanics of a bonding curve are relatively straightforward. The curve can be represented as a single smart contract with options for purchasing and selling the underlying token. The token to be sold can have either an uncapped supply with the bonding curve as an authorized minter or a predetermined maximum supply that is escrowed in the bonding curve contract. As users purchase the token, the bonding curve escrows the incoming funding for the point in the future when a user may want to sell back against the curve.

The growth rate of the bonding curve is important in determining users' performance. A linear growth rate would generously reward early users if the token grows to a sufficiently large supply. An even more extreme return could result from a superlinear growth rate, such as $TKN = S^2$. The first token would cost 1 ETH and the 100th would cost 10,000 ETH. In practice, most projects would use a sublinear growth rate or a logistic function that converges on an upper bounded price.

Super Linear Bonding Curve

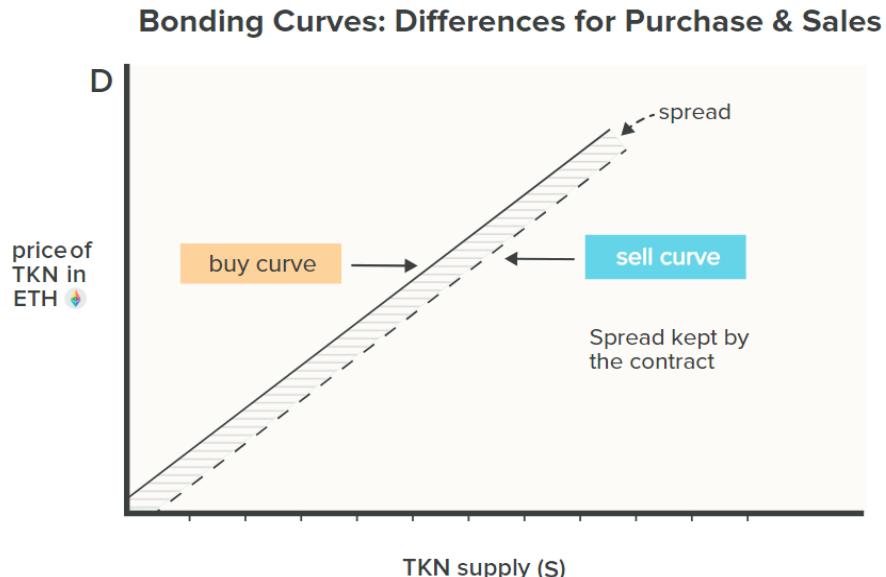


Logistic/Sigmoid Bonding Curves



A bonding curve can have a different price curve for buyers and for sellers. The selling curve could have a lower growth rate or intercept than the buying curve. The spread between the curves would be the value (in this case ETH) accrued to the smart contract and could represent a fee for usage or used to finance more-complex functionality within the system. As long as the contract

maintains sufficient collateral to sell back down the entire sell curve, the contract is capitalized and able to fulfill any sell demand.



4.6 Incentives

Incentives within cryptoeconomic systems including DeFi are extremely important in encouraging desired (positive incentive) and discouraging undesired (negative incentive) user behaviors. The term *incentive* is quite broad, but we narrow our discussion to direct token payments or fees. We will look at two different categories of incentives: *staked incentives* and *direct incentives*. Staked incentives apply to a balance of tokens custodied in a smart contract. Direct incentives apply to users within the system who do not have a custodied balance.

Mechanisms in the contract determine the source of any reward funds and the destination for fees. Reward funds can be issued through inflation or by minting (Section 4.5.2) as well as custodied in the smart contract (Section 4.4). Funds removed as a fee can be burned (Section 4.5.1) or can be retained in the smart contract's custody. Reward funds can also be issued as a direct incentive to the platform's participants or can be raised through an auction to repay a debt. A mechanism might instigate a burn to reduce the supply of a particular token in order to increase price pressure.

4.6.1 Staking Rewards

A *staking reward* is a positive staked incentive by which a user receives a bonus in his token balance based on the stake he has in the system. Several verticals of incentive customization are possible:

- Stake requirement options:
 - minimum threshold or applied to all staked balances on a pro rata basis
- Reward options:
 - Fixed payout or pro rata payout
 - Same token type as staked or a distinct token

The Compound protocol (Section 6.1.2) issues staking rewards on user balances that are custodied in a borrowing or lending position. These rewards are paid in a separate token (COMP) funded by custodied COMP, which has a fixed supply, and applied to all staked balances on a pro rata basis. The Synthetix protocol (Section 6.3.3) issues staking rewards on staked SNX, its protocol token which has unlimited supply. The rewards are paid in SNX, funded by inflation, and issued only if the user meets a minimum-collateralization-ratio threshold.

4.6.2 Slashing (Staking Penalties)

Slashing is the removal of a portion of a user's staked balance, thereby creating a negative staked incentive. Slashing occurs as the result of an undesirable event. A *slashing condition* is a mechanism that triggers a slashing. As with staking rewards, several verticals of slashing customization are possible:

- Removed funds options:
 - Complete or partial slashing
- Slashing condition options:
 - Undercollateralization triggers liquidation
 - Detectable malicious behavior by user
 - Change in market conditions triggers necessary contraction

In Section 4.8 on collateralized loans, we will illustrate the common slashing mechanism of *liquidation*. In a liquidation, potential liquidators receive a direct incentive to execute the liquidation through auctioning or directly selling the collateral; the balance of funds remaining after the liquidation stays with the original owner. An example of slashing due to market changes not related to debt is an algorithmic stablecoin. This system might directly reduce a user's token balance when the price depreciates in order to return the supply-weighted price to, say, \$1.

4.6.3 Direct Rewards and Keepers

Direct rewards are positive incentives that include payments or fees associated with user actions. As we mentioned in Section 4.1, all Ethereum interactions begin with a transaction, and all transactions begin with an externally owned account. An EOA, whether controlled by a human user or an off-chain bot, is (importantly) off chain, and thus autonomous monitoring of market conditions is either expensive (costs gas) or technically infeasible. As a result, no transaction happens automatically on Ethereum without being purposely set in motion.

The classic example of a transaction that must be set in motion is when a collateralized debt position becomes undercollateralized. This use case does not automatically trigger a liquidation; the EOA must trigger the liquidation. For this use case and others, EOAs generally receive a direct incentive to trigger the contract. The contract then evaluates the conditions and liquidates or updates if everything is as expected. We will discuss the mechanics of liquidation in more depth in Section 4.8 on collateralized loans.

A *keeper* is a class of EOA incentivized to perform an action in a DeFi protocol or other dApp. A keeper is rewarded by receiving a fee, either flat or percentage of the incented action. With sufficient incentivization, autonomous monitoring can be outsourced off chain, thus creating robust economies and new profit opportunities. Keeper rewards may also be structured as an auction to ensure competition and best price. Keeper auctions are very competitive because the information available in the system is almost entirely public. A side effect of direct rewards for keepers is that gas prices can inflate due to the competition for these rewards. That is, more keeper activity generates additional demand for transactions, which in turn increases the price of gas.

4.6.4 Fees

Fees are typically a funding mechanism for the features of the system or platform. They can be flat or percentage based, depending on the desired incentive. Fees can be imposed as a direct negative incentive or can be accrued on staked balances. Accrued fees must have an associated staked balance to ensure the user pays them. Because of the pseudonymous anonymous nature of Ethereum accounts—all that is known about an Ethereum user is his wallet balance and interactions with various contracts on Ethereum—the imposition of fees is a design challenge. If the smart contract is open to any Ethereum account, the only way to guarantee off-chain enforcement or legal intervention is for all debts to be backed by staked collateral, which is transparent and enforceable. The challenges created by anonymity make other mechanisms, such as reputation, unsuitable alternatives to staked balances.

4.7 Swap

A swap is simply the exchange of one type of token to another. The key benefit of swapping in DeFi is that it is atomic and noncustodial. Funds can be custodied in a smart contract with withdrawal rights that can be exercised at any time before the swap is completed. If the swap does not complete, all parties involved retain their custodied funds. The swap only executes when the exchange conditions are agreed to and met by all parties, and are enforced by the smart contract. If any condition is not met, the entire transaction is cancelled. A platform that facilitates token swapping on Ethereum in a noncustodial fashion is a *decentralized exchange* (DEX). There are two primary mechanisms for DEX liquidity: one is an order-matching approach and the other is an *Automated Market Maker*.

4.7.1 Order Book Matching

Order-book matching is a system in which all parties must agree on the swap exchange rate. Market makers can post bids and asks to a DEX and allow takers to fill the quotes at the pre-

agreed-upon price. Until the offer is taken, the market maker retains the right to remove the offer or update the exchange rate as market conditions change. A leading example of a fully on-chain order book is [Kyber](#).

The order-matching approach is expensive and inefficient because each update requires an on-chain transaction. An insurmountable inefficiency with an order-book matching is that both counterparties must be willing and able to exchange at the agreed-upon rate for the trade to execute. This requirement creates limitations for many smart contract applications in which demand for exchange liquidity cannot be dependent on a counterparty's availability. An innovative alternative is an Automated Market Maker.

4.7.2 Automated Market Makers (AMMs)

An Automated Market Maker (AMM) is a smart contract that holds assets on both sides of a trading pair and continuously quotes a price for buying and for selling. Based on executed purchases and sales, the contract updates the asset size behind the bid and the ask and uses this ratio to define its pricing function. The contract can also take into account more complex data than relative bid/ask size when determining price. From the contract's perspective, the price should be risk-neutral where it is indifferent to buying or selling.

A naive AMM might set a fixed price ratio between two assets. With a fixed price ratio, when the market price shifts between the assets, the more valuable asset would be drained from the AMM and arbitrated on another exchange where trading is occurring at the market price. The AMM should have a pricing function that can converge on the market price of an asset so that it becomes more expensive to purchase an asset from the trading pair as the ratio of that asset to the others in the contract decreases.

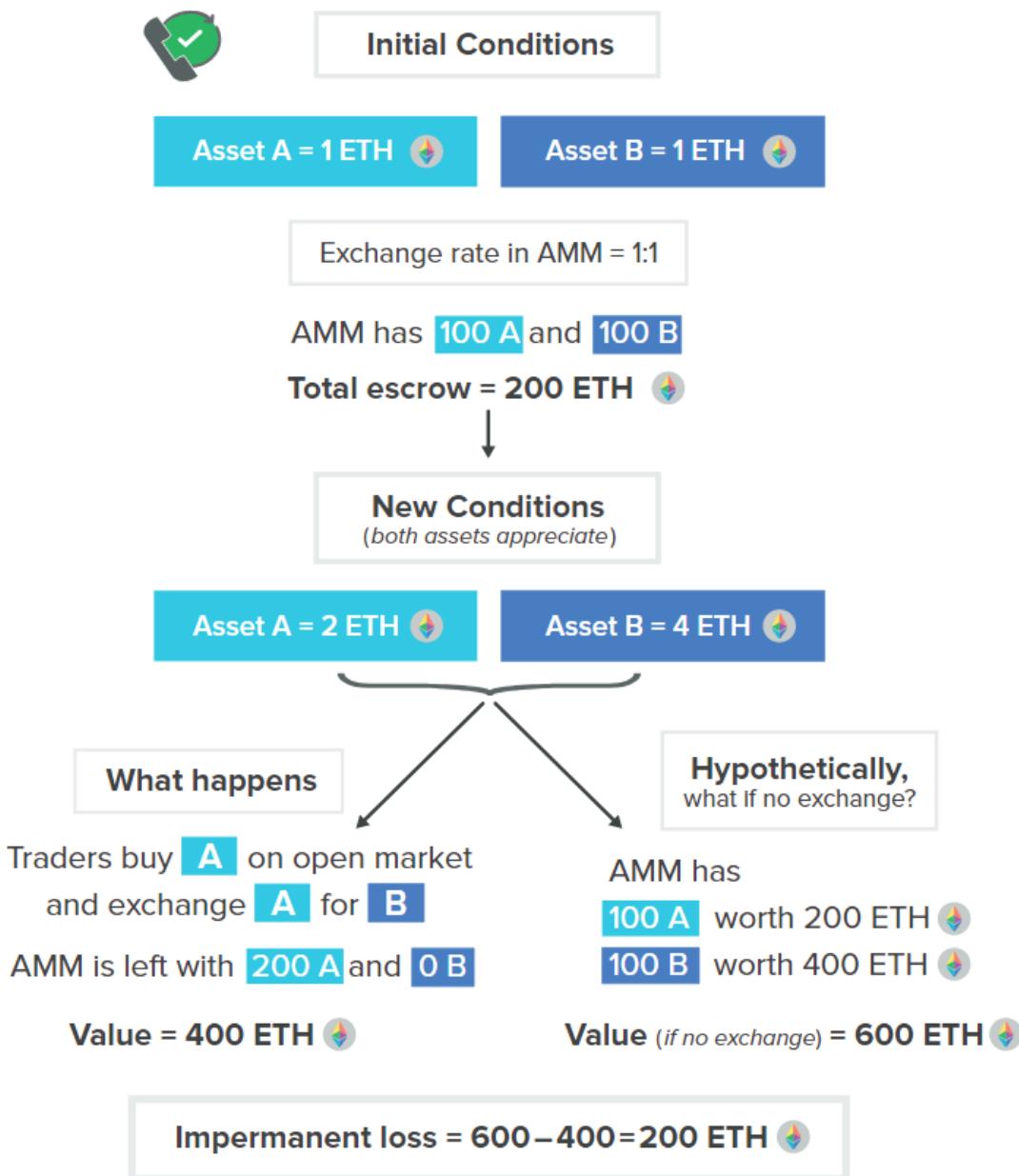
The major benefits of an AMM are constant availability and that a traditional counterparty is not necessary to execute a trade. These provisions are very important for smart contracts and DeFi development because of the guarantee that a user can exchange assets at any moment if necessary. A user maintains custody of her funds until she completes the trade, hence, counterparty risk is zero. An additional benefit is *composable liquidity*, which means any exchange contract can plug into the liquidity and exchange rates of any other exchange contract. AMMs make this particularly easy because of their guaranteed availability and their allowing one-sided trading against the contract. Composable liquidity correlates comfortably with the concept of DeFi Legos.

One drawback to an AMM is the concept of *impermanent loss*, the opportunity-cost dynamic between offering assets for exchange and holding the underlying assets to potentially profit from the price movement. The loss is impermanent because it can be recovered if the price reverts to its original level. To illustrate, consider two assets, A and B, each initially worth 1 ETH as in Exhibit X. The AMM contract holds identical quantities of 100 of each asset and naively offers both at a fixed exchange rate of 1:1. We use ETH as the unit of account to track the contract's return on its holdings and any impermanent loss. At the given balances and market exchange rates, the contract has 200 ETH in escrow. Suppose asset B's price appreciates to 4 ETH in the wider market and asset A's price appreciates to 2 ETH. Arbitrageurs exchange all of asset B in the

contract for asset A because asset B is more valuable. The contract then holds 200 of asset A worth 400 ETH. In this case, the contract's real return is 100%.

If, however, the contract does not sell asset B, the contract's value would be 600 ETH. The contract has an impermanent loss equal to 200 ETH, the difference between 600 ETH and 400 ETH. If the contract's holdings return to parity between assets A and B, the impermanent loss disappears. If the goal for liquidity held in the contract is profit, any fees charged must exceed the amount of the impermanent loss.

AUTOMATED MARKET MAKER



Impermanent loss occurs for any shift in price and liquidity, because the contract is structured to sell the appreciating asset and to buy the depreciating asset. An important feature of impermanent loss is *path independence*. In our example, it is irrelevant whether 1 or 100 traders consumed all the liquidity. The final exchange rate and contract asset ratios yield the same impermanent loss regardless of the number of trades or the direction of the trades. Because of path independence, impermanent loss is minimized on trading pairs that have correlated prices (*mean-reverting pairs*). Thus, stablecoin trading pairs are particularly attractive for AMMs.

4.8 Collateralized Loans

Debt and lending are perhaps the most important financial mechanisms that exist in DeFi, and more generally, in traditional finance. On the one hand, these mechanisms are a powerful tool for efficiently allocating capital, increasing return-bearing risk exposure, and expanding economic growth. On the other hand, excess debt in the system can cause instability, leading to large economic and market contractions. These benefits and risks are amplified in DeFi, because the counterparties share an adversarial and integrated environment. Platforms are increasingly interdependent, and a debt-fueled collapse in one part of the system can quickly contaminate all connected protocols—and expand outward.

Any loan of non-zero duration (e.g., foreshadowing flash loan, as we discuss in Section 4.9) must be backed by an equivalent or excess amount of collateral. Requiring collateral contractually prevents a counterparty from defaulting. An uncollateralized mechanism raises the risk that a counterparty could steal funds, especially in an open and anonymous system such as Ethereum. A risk of overcollateralized positions is that the collateral becomes less valuable than the debt, leading to a foreclosure without an option for recovery. Therefore, more-volatile types of collateral require larger collateralization ratios in order to mitigate this risk.

We have already mentioned the mechanism of liquidation and now we will explain it in detail. To avoid liquidation it is imperative that debt remain overcollateralized by a margin sufficiently large that moderate price volatility does not place the collateral value in jeopardy. Smart contracts commonly define a minimum collateralization threshold below which the collateral can be liquidated and the position closed. The collateral could be auctioned or directly sold on a DEX, likely with an AMM, at the market price.

As stated in Section 4.6, positions in the Ethereum blockchain cannot be liquidated automatically, so an incentive is needed. The incentive often takes the form of a percentage fee allocated to an external keeper who is able to liquidate the position and collect the reward. Any remaining collateral is left to the original holder of the position. In some cases, the system will leave all remaining collateral to the keeper as a stronger incentive. Because the penalty for liquidation is high and most collateral types are volatile, platforms generally allow users to top up their collateral to maintain healthy collateralization ratios.

An interesting implication of collateralized loans and token supply adjustment (Section 4.5) is that collateralization can back the value of a synthetic token. The synthetic token is an asset created and funded by a debt, which is the requirement to repay the synthetic token in order to

reclaim the collateral. The synthetic token can have a utility mechanism or represent a complex financial derivative, such as an option or bond (e.g., Synthetix Synth, discussed in Section 6.3.3 and Yield yToken, discussed in Section 6.3.1). A stablecoin that tracks the price of an underlying asset can also be a synthetic token of this type (e.g., MakerDAO DAI, discussed in Section 6.1.1).

4.9 Flash Loan (Uncollateralized Loan)

A financial primitive that uniquely exists in DeFi and dramatically broadens certain types of financial access is a *flash loan*. In traditional finance, a loan is an instrument designed to efficiently allocate excess capital from a person or entity who wishes to employ it (lender) to a person or entity who needs capital to fund a project or to consume (borrower). A lender is compensated for providing the capital and bearing the risk of default by the interest amount charged over the life of the loan. The interest rate is typically higher the longer the duration of the loan, because the longer time to repay exposes the lender to greater risk that the borrower may default.

Reversing the concept leads to the conclusion that shorter-term loans should be less risky and therefore require less compensation for the lender. A flash loan is an instantaneous loan paid back within the same transaction. A flash loan is similar to an overnight loan in traditional finance, but with a crucial difference—repayment is required within the transaction and enforced by the smart contract.

A thorough understanding of an Ethereum transaction is important for understanding how flash loans work. One clause in the transaction is vital: if the loan is not repaid with required interest by the end of the transaction, the whole process reverts to the state before any money ever left the lender's account. In other words, either the user successfully employs the loan for the desired use case and completely repays it in the transaction or the transaction fails and everything resets as if the user had not borrowed any money.

Flash loans essentially have zero counterparty risk or duration risk. However, there is always smart contract risk (e.g., a flaw in the contract design, see Section 7.1). They allow a user to take advantage of arbitrage opportunities or refinance loans without pledging collateral. This capability allows anyone in the world to have access to opportunities that typically require very large amounts of capital investment. In time, we will see similar innovations that could not exist in the world of traditional finance.

5. Problems DeFi Solves

In this chapter, we will address the concrete solutions that DeFi presents to the five flaws of traditional finance: inefficiency, limited access, opacity, centralized control, and lack of interoperability.

5.1 Inefficiency

The first of the five flaws of traditional finance is inefficiency. DeFi can accomplish financial transactions with high volumes of assets and low friction that would generally be a large organizational burden for traditional finance. DeFi creates reusable smart contracts in the form of dApps designed to execute a specific financial operation. These dApps are available to any user who seeks that particular type of service, for example, to execute a put option, regardless of the size of the transaction. A user can largely self-serve within the parameters of the smart contract and of the blockchain the application lives on. In the case of Ethereum-based DeFi, the contracts can be used by anyone who pays the flat gas fee, usually around \$15.00 for a transfer and \$25.00 for a dApp feature such as leveraging against collateral. Once deployed, these contracts continually provide their service with near-zero organizational overhead.

5.1.1 Keepers

We introduced the concept of a keeper in Section 4.6.3. Keepers are external participants directly incentivized to provide a service to DeFi protocols, such as monitoring positions to safeguard that they are sufficiently collateralized or triggering state updates for various functions. To ensure that a dApp's benefits and services are optimally priced, keeper rewards are often structured as an auction. Pure, open competition provides value to DeFi platforms by guaranteeing users pay the market price for the services they need.

5.1.2 Forking

Another concept that also incentivizes efficiency is a *fork*. A fork, in the context of open source code, is a copy and reuse of the code with upgrades or enhancements layered on top. A common fork of blockchain protocols is to reference them in two parallel currencies and chains. Doing so creates competition at the protocol level and creates the best possible smart contract platform. Not only is the code of the entire Ethereum blockchain public and forkable, but each DeFi dApp built on top of Ethereum is as well. Should inefficient or suboptimal DeFi applications exist, the code can be easily copied, improved, and redeployed through forking. Forking and its benefits arise from the open nature of DeFi and blockchains.

Forking creates an interesting challenge to DeFi platforms, namely, *vampirism*. Vampirism is an exact or near-carbon copy of a DeFi platform designed to poach liquidity or users by offering larger incentives than the platform it is copying. The larger incentives usually take the form of inflationary rewards offered at a far higher rate than the original platform offers. Users might be attracted to the higher potential reward for the same functionality, which would cause a reduction in usage and liquidity on the initial platform.

If the inflationary rewards are flawed, over long-term use the clone could perhaps collapse after a large asset bubble. The clones could also select closer to optimal models and replace the original platform. Vampirism is not an inherent risk or flaw, but rather a complicating factor arising from the pure competition and openness of DeFi. The selection process will eventually give rise to more robust financial infrastructure with optimal efficiency.

5.2 Limited Access

As smart contract platforms move to more-scalable implementations, user friction falls, enabling a wide range of users, and thus mitigates the second flaw of traditional finance: limited access. DeFi gives large underserved groups, such as the global population of the unbanked as well as small businesses that employ substantial portions of the workforce (for example, nearly 50% in the United States) direct access to financial services. The resulting impact on the entire global economy should be strongly positive. Even consumers who have access to financial services in traditional finance, such as bank accounts, mortgages, and credit cards, do not have access to the financial products with the most competitive pricing and most favorable terms; these products and structures are restricted to large institutions. DeFi allows any user access to the entirety of its financial infrastructure, regardless of her wealth or geographic location.

5.2.1 Yield Farming

Yield farming, mentioned in Section 4.5.2, provides access to many who need financial services but whom traditional finance leaves behind. To summarize, yield farming provides inflationary or contract-funded rewards to users for staking capital or utilizing a protocol. These rewards are payable in the same underlying asset the user holds or in a distinct asset such as a governance token. Any user can participate in yield farming. A user can stake an amount of any size, regardless of how small, and receive a proportional reward. This capability is particularly powerful in the case of governance tokens. A user of a protocol that issues a governance token via yield farming becomes a partial owner of the platform through the issued token. A rare occurrence in traditional finance, this process is a common and celebrated way to give ownership of the platform to the people who use and benefit from it.

5.2.2 Initial DeFi Offering

An interesting consequence of yield farming is that a user can create an *Initial DeFi Offering* (IDO) by market making his own Uniswap (section 6.2.1) trading pair. The user can set the initial exchange rate by becoming the first liquidity provider on the pair. Suppose the user's token is called DFT and has a total supply of 2 million. The user can make each DFT worth 0.10 USDC by opening the market with 1 million DFT and 100,000 USDC. Any ERC-20 token holder can purchase DFT, which drives up the price. As the only liquidity provider, the user also receives all of the trading fees. In this way, the user is able to get his token immediate access to as many users as possible. The method sets an artificial price floor for the token if the user controls the supply outside of the amount supplied to the Uniswap market, and as such, inhibits price discovery. The trade-offs of an IDO should be weighed as an option, or strategy, for a user's token distribution.

IDOs democratize access to DeFi in two ways. First, an IDO allows a project to list on high-traffic DeFi exchanges that do not have barriers to entry beyond the initial capital. Second, an IDO allows a user access to the best new projects immediately after the project lists.

5.3 Opacity

The third drawback of traditional finance is opacity. DeFi elegantly solves this problem through the open and contractual nature of agreements. We will explore how smart contracts and tokenization improve transparency within DeFi.

5.3.1 Smart Contracts

Smart contracts provide an immediate benefit in terms of transparency. All parties are aware of the capitalization of their counterparties and, to the extent required, can see how funds will be deployed. The parties can read the contracts themselves to determine if the terms are agreeable to eliminate any ambiguity as to what will happen when they interact under the contract terms. This transparency substantially eases the threat of legal burdens and brings peace of mind to smaller players who, in the current environment of traditional finance, could be abused by powerful counterparties through delaying or even completely withholding their end of a financial agreement. Realistically, the average consumer does not understand the contract code, but can rely on the open-source nature of the platform and the wisdom of the crowd to feel secure. Overall, DeFi mitigates counterparty risk and thus creates a host of efficiencies not present under traditional finance.

DeFi participants are accountable for acting in accordance with the terms of the contracts they use. One mechanism for ensuring the appropriate behavior is staking. Staking is escrowing a cryptoasset into a contract, so that the contract releases the cryptoasset to the appropriate counterparty only after the contract terms are met; otherwise, the asset reverts to the original holder. Parties can be required to stake on any claims or interactions they make. Staking enforces agreements by imposing a tangible penalty for the misbehaving side and a tangible reward for the counterparty. The tangible reward should be as good as or even better than the outcome of the original terms of the contract. These transparent incentive structures provide much securer and more obvious guarantees than traditional financial agreements.

Another type of smart contract in DeFi that improves transparency is a token contract. Tokenization allows for transparent ownership and economics within a system. Users can know exactly how many tokens are in the system as well as the inflation and deflation parameters.

5.4 Centralized Control

The fourth flaw of traditional finance is the strong control exerted by governments and large institutions that hold a virtual monopoly over elements such as the money supply, rate of inflation, and access to the best investment opportunities. DeFi upends this centralized control by relinquishing control to open protocols having transparent and immutable properties. The community of stakeholders or even a predetermined algorithm can control a parameter, such as the inflation rate, of a DeFi dApp. If a dApp contains special privileges for an administrator, all users are aware of the privileges, and any user can readily create a less-centralized counterpart.

The open-source ethos of blockchain and the public nature of all smart contracts assures that flaws and inefficiencies in a DeFi project can be readily identified and “forked away” by users who copy and improve the flawed project. Consequently, DeFi strives to design protocols that naturally and elegantly incentivize stakeholders and maintain a healthy equilibrium through careful mechanism design. Naturally, trade-offs exist between having a centralized party and not having one. Centralized control allows for radically decisive action in a crisis, sometimes the necessary approach but also perhaps an overreaction. The path to decentralizing finance will certainly encounter growing pains because of the challenges in pre-planning for every eventuality and economic nuance. Ultimately, however, the transparency and security gained through a decentralized approach will lead to strong robust protocols that can become trusted financial infrastructure for a global user base.

5.4.1 Decentralized Autonomous Organization

A *decentralized autonomous organization (DAO)* has its rules of operation encoded in smart contracts that determine who can execute what behavior or upgrade. It is common for a DAO to have some kind of *governance token*, which gives an owner some percentage of the vote on future outcomes. We will explore governance in much more detail later.

5.5 Lack of Interoperability

We will now touch on the lack-of-interoperability aspect of traditional finance that DeFi solves. Traditional financial products are difficult to integrate with each other, generally requiring at minimum a wire transfer, but in many cases cannot be recombined. The possibilities for DeFi are substantial and new innovations continue to grow at a non-linear rate. This growth is fueled by the ease of composability of DeFi products. Once one has some base infrastructure to, for example, create a synthetic asset, any new protocols allowing for borrowing and lending can be applied. A higher layer would allow for attainment of leverage on top of borrowed assets. Such composability can continue in an increasing number of directions as new platforms arise. For this reason, *DeFi Legos* is an analogy often used to describe the act of combining existing protocols into a new protocol. We will discuss below a few advantages to this composability, namely tokenization and networked liquidity.

5.5.1 Tokenization

Tokenization is a critical way in which DeFi platforms integrate with each other. Take for example a percentage ownership stake in a private commercial real estate venture. In traditional finance to use this asset as collateral for a loan or as margin to open a levered derivative position would be quite difficult. Because DeFi relies on shared interfaces, applications can directly plug into each other’s assets, repackage, and subdivide positions as needed. DeFi has the potential to unlock liquidity in traditionally illiquid assets through tokenization. A simple use case would be creating fractional shares from a unitary asset such as a stock. We can extend this concept to give fractional ownership to scarce resources such as rare art. The tokens can be used as collateral for any other DeFi service, such as leverage or derivatives.

We are able to invert this paradigm to create token bundles of groups of real-world or digital assets and trade them like an ETF. Imagine a dApp similar to a real estate investment trust (REIT), but with the added capability of allowing the owner to subdivide the REIT into the individual real estate components to select a preferred geographic distribution and allocation within the REIT. Ownership of the token provides direct ownership of the distribution of the properties. The owner can trade the token on a decentralized exchange to liquidate the position.

Tokenizing hard assets, such as real estate or precious metals, is more difficult than tokenizing digital assets because the practical considerations related to the hard assets, such as maintenance and storage, cannot be enforced by code. Legal restrictions across jurisdictions are also a challenge for tokenization; nevertheless, the utility of secure, contractual tokenization for most use cases should not be underestimated.

A tokenized version of a position in a DeFi platform is a pluggable derivative asset that is usable in another platform. Tokenization allows the benefits and features of one position to be portable. The archetypal example of portability through tokenization is Compound, which we will discuss in Section 6.2. Compound allows for robust lending markets in which a position can accrue variable-rate interest denominated in a given token, and the position itself is a token. If, for example, the base asset is ETH, the ETH deposit wrapper known as cETH (cToken) can be used in place of the base asset. The result is an ETH-backed derivative that is also accruing variable-rate interest per the Compound protocol. Tokenization, therefore, unlocks new revenue models for dApps because they can plug asset holdings directly into Compound or use the cToken interface to gain the benefits of Compound's interest rates.

5.5.2 Networked Liquidity

The concept of interoperability extends easily to liquidity in the exchange use case. Traditional exchanges, in particular those that retail investors typically use, cannot readily share liquidity with other exchanges without special access to a prime broker, which is generally limited to hedge funds. In DeFi, as a subcomponent of the contract, any exchange application can leverage the liquidity and rates of any other exchange on the same blockchain. This capability allows for networked liquidity and leads to very competitive rates for users within the same application.

6. DeFi Deep Dive

DeFi can be loosely broken into sectors based on the functionality type of the dApp. Many dApps could fit into multiple categories, so we attempt to place them into the most relevant category. We examine DeFi platforms in the taxonomy of lending/credit facilities, DEXes, derivatives, and tokenization.¹⁷ We mainly focus on the Ethereum network due to its popularity, but DeFi

¹⁷ A large number of DeFi resources are available. For example, see <https://defipulse.com/defi-list/> and <https://github.com/ong/awesome-decentralized-finance>. We do not cover all applications. For example, insurance is a growing area in DeFi that offers to reinvent traditional insurance markets.

innovations are occurring on many blockchains including [Stellar](#) and [EOS](#). [Polkadot](#) is another platform that employs a type of Proof of Stake consensus.

6.1 Credit/Lending

6.1.1 MakerDAO

[MakerDAO](#) (DAO is decentralized autonomous organization) is often considered an exemplar of DeFi. In order for a series of applications to build on each other, there must necessarily be a foundation. The primary value-add of MakerDAO is the creation of a crypto-collateralized stablecoin, pegged to USD. This means the system can run completely from within the Ethereum blockchain without relying on outside centralized institutions to back, vault and audit the stablecoin. MakerDAO is a two-token model where a governance token MKR yields voting rights on the platform and participates in value capture. The second token is the stablecoin, called DAI, and is a staple token in the DeFi ecosystem with which many protocols integrate - including a few we will discuss later.

DAI is generated as follows. A user can deposit ETH or other supported ERC-20 assets into a *Vault*. A Vault is a smart contract that escrows collateral and keeps track of the USD-denominated value of the collateral. The user can then mint DAI up to a certain collateralization ratio on their assets. This creates a “debt” in DAI that must be paid back by the Vault holder. The DAI is the corresponding asset that can be used any way the Vault holder wishes. For example, the user can sell the DAI for cash or lever it into more of the collateral asset,¹⁸ and repeat the process. Due to the volatility of ETH and most collateral types, the collateralization requirement is far in excess of 100% and usually in the 150-200% range.

The basic idea is not new; it is simply a collateralized debt position. For example, a homeowner in need of some liquidity can pledge their house as collateral to a bank and receive a mortgage loan structured to include a cash takeout. The price volatility of ETH is much greater than for a house and, as such, collateralization ratios for the ETH-DAI contract are higher. In addition, no centralized institution is necessary as everything happens within the Ethereum blockchain.

Let's consider a simple example. Suppose an ETH owner needs liquidity but does not want to sell her ETH because she thinks it will appreciate. The situation is analogous to the homeowner who needs liquidity but does not want to sell her house. Let's say an investor has 5 ETH at a market price of \$200 (total value of \$1,000). If the collateralization requirement is 150%, then the investor can mint up to 667 DAI (\$1,000/1.5 with rounding). The collateralization ratio is set high to reduce the probability that the loan debt exceeds the collateral value, and for the DAI token to be credibly pegged to the USD, the system needs to avoid the risk that the collateral is worth less than \$1=1 DAI.

¹⁸ It is possible to deposit ETH into the contract and receive DAI. An investor could use that DAI to buy more ETH and repeat the process, allowing the investor to create a leveraged ETH position.

Given the collateralization ratio of 1.5, it would be unwise to mint the 667 DAI because if the ETH ever dropped below \$200, the contract would be undercollateralized, the equivalent of a “margin call”. We are using traditional finance parlance, but in DeFi there is no communication from your broker about the need to post additional margin or to liquidate the position and also no grace period. Liquidation can happen immediately.

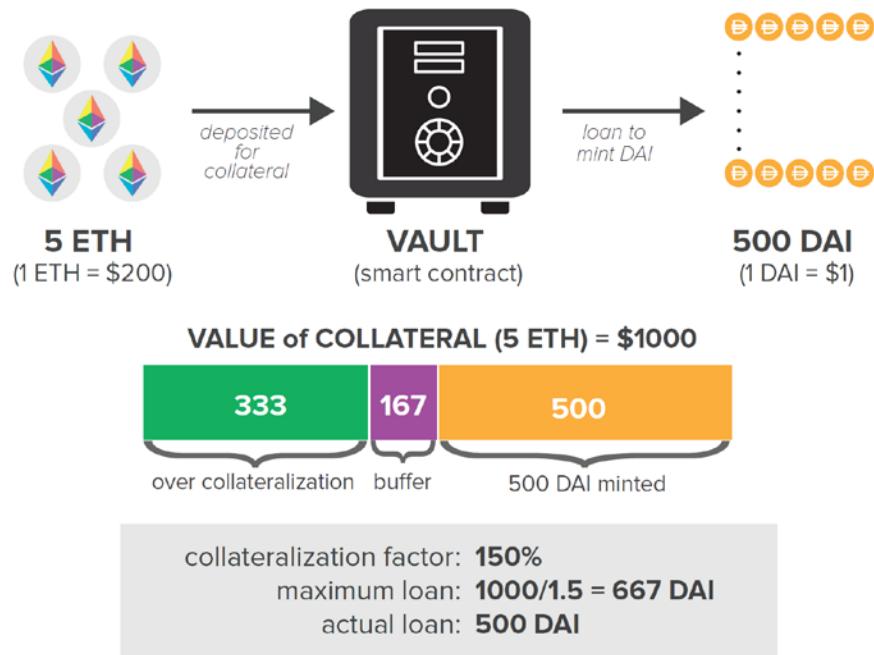
As such, most investors choose to mint less than 667 DAI to give themselves a buffer. Suppose the investor mints 500 DAI, which implies a collateralization ratio of 2.0 ($\$1,000/2.0 = 500$). Let’s explore two scenarios. First, suppose the price of ETH rises by 50% so that the collateral is worth \$1,500. Now, the investor can increase the size of his loan. To maintain the collateralization of 200%, the investor can mint an extra 250 DAI.

A more interesting scenario is when the value of the collateral drops. Suppose the value of the ETH drops by 25% from \$200 to \$150. In this case, the value of the collateral drops to \$750 and the collateralization ratio drops to 1.5 ($\$750/1.5 = 500$).

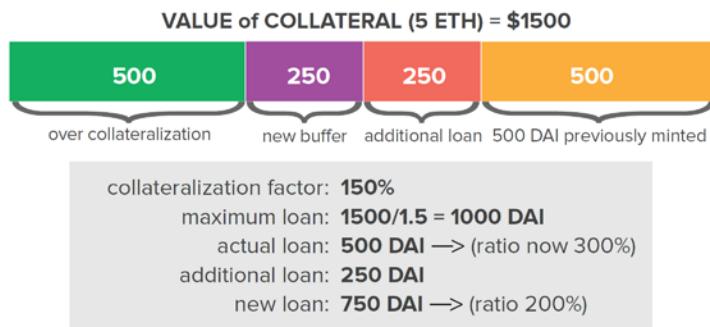
The Vault holder faces three scenarios. First, he can increase the amount of collateral in the contract (by, for example, adding 1 ETH). Second, he can use the 500 DAI to pay back the loan and repatriate the 5 ETH. These ETH are now worth \$250 less, but the depreciation in value would have happened irrespective of the loan. Third, the loan is liquidated by a *keeper* (any external actor). A keeper is incentivized to find contracts eligible for liquidation. The keeper auctions the ETH for enough DAI to pay off the loan. In this case, 3.33 ETH would be sold and 1.47 would be returned to the Vault holder (the keeper earns an incentive fee of 0.2 ETH). The Vault holder then has 500 DAI worth \$500 and 1.47 ETH worth \$220. This analysis does not include gas fees.

Two forces in this process reinforce the stability of DAI. The first is the overcollateralization. The second is the market actions. In the liquidation, ETH are sold and DAI are purchased, which exerts positive price pressure on DAI. This simple example does not address many features in the MakerDAO ecosystem, in particular, the fee mechanisms and the debt limit, which we will now explore.

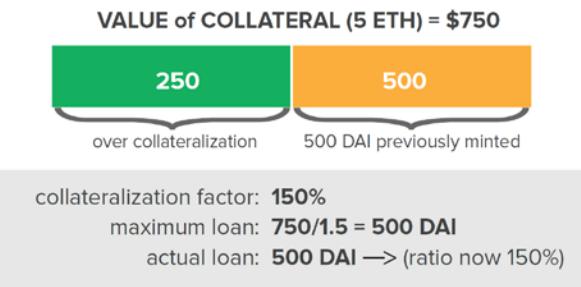
EXHIBIT A



Scenario 1 ETH appreciates 50% \$200 → \$300



Scenario 2 ETH depreciates 25% \$200 → \$150





The viability of the MakerDAO ecosystem critically depends on DAI maintaining a 1:1 peg to the USD. Various mechanisms are in place to incentivize demand and supply in order to drive the price toward the peg. The primary mechanisms for maintaining the peg are the debt ceiling, stability fee, and DAI Savings Rate (DSR). These parameters are controlled by holders of the governance token Maker (MKR) and MakerDAO governance, which we will discuss toward the end of this section.

The Stability Fee is a variable interest rate paid in DAI by Vault holders on any DAI debt they generate. The interest rate can be raised or lowered (even to a negative value) to incentivize the generation or repayment of DAI to drive its price toward the peg. The Stability Fee funds the DSR, a variable rate any DAI holder can earn on their DAI deposit. The DSR compounds on a per-block basis. The Stability Fee, which must always be greater or equal to the DSR, is enforced by the smart contracts powering the platform. Lastly, a smart contract–enforced DAI debt ceiling can be adjusted to allow for more or less supply to meet the current level of demand. If the protocol is at the debt ceiling, no new DAI is able to be minted in new Vaults until the old debt is paid or the ceiling is raised.

To stay above the liquidation threshold, a user can deposit more collateral into the Vault to keep the DAI safely collateralized. When a position is deemed to be under the liquidation ratio, a keeper can initiate an auction (sell some of the ETH collateral¹⁹) to liquidate the position and close the Vault holder's debt. The *Liquidation Penalty* is calculated as a percentage of the debt and is deducted from the collateral in addition to the amount needed to close the position.

After the auction, any remaining collateral reverts to the Vault owner. The Liquidation Penalty acts as an incentive for market participants to monitor the Vaults and trigger an auction when a position becomes undercollateralized. If the collateral drops so far in value that the DAI debt cannot be fully repaid, the position is closed, and the protocol accrues what is known as *Protocol Debt*. A buffer pool of DAI exists to cover Protocol Debt, but in certain circumstances the debt can be too great for even the buffer pool to cover. The solution involves the governance token MKR and the governance system.

¹⁹ The amount of ETH available for sale depends on the collateralization. Any unneeded collateral remains in the contract for the Vault holder to withdraw.

The MKR token controls MakerDAO. Holders of the token have the right to vote on protocol upgrades, including supporting new collateral types and tweaking parameters such as collateralization ratios. MKR holders are expected to make decisions in the best financial interest of the platform. Their incentive is that a healthy platform should increase the value of their share in the platform's governance. For example, poor governance could lead to a situation as described earlier in which the buffer pool is not sufficient to pay back the Protocol Debt. In this case, newly minted MKR tokens are auctioned off in exchange for DAI and the DAI are used to pay back the Protocol Debt. This process is *Global Settlement*, a safety mechanism intended for use only when all other measures have failed. Global Settlement dilutes the MKR share, which is why stakeholders are incentivized to avoid it and keep Protocol Debt to a minimum.

MKR holders are collectively the owners of the future of MakerDAO. A proposal and corresponding approved vote can change any of the parameters available on the platform. Other possible parameter changes include supporting new collateral types for Vaults and adding upgrades to functionality. MKR holders could for instance vote to pay themselves a dividend funded by the spread between the interest payments paid by Vault holders and the DAI Savings Rate. The reward of receiving this dividend would need to be weighed against any negative community response (e.g., a backlash against rent seeking from a previously no-rent protocol) that might decrease the value of the protocol and the MKR token.

A number of features make DAI attractive to users. Importantly, users can purchase and utilize DAI without having to go through the process of generating it in a Vault—they can simply purchase DAI on an exchange. Therefore, users do not need to know the underlying mechanics of how DAI are created. Holders can easily earn the DAI Savings Rate by using the protocol. More technologically and financially sophisticated users can use the MakerDAO web portal to generate Vaults and create DAI to get liquidity from their assets without having to sell them. It is easy to sell DAI and purchase an additional amount of the collateral asset to get leverage.

A noteworthy drawback to DAI is that its supply is always constrained by demand for ETH-collateralized debt. No clear arbitrage loop exists to maintain the peg. For example, the stablecoin USDC is always redeemable by Coinbase for \$1, with no fees. Arbitrageurs have a guaranteed (assuming solvency of Coinbase) strategy in which they can buy USDC at a discount or sell it at a premium elsewhere and redeem on Coinbase. This is not true for DAI. Irrespective of any drawbacks, the simplicity of DAI makes it an essential building block for other DeFi applications.

Traditional Finance Problem	MakerDAO Solution
<i>Centralized Control:</i> Interest rates are influenced by the US Federal Reserve and access to loan products controlled by regulation and institutional policies.	MakerDAO platform is openly controlled by the MKR holders.
<i>Limited Access:</i> Obtaining loans is difficult for a large majority of the population.	Open ability to take out DAI liquidity against an overcollateralized position in any supported ERC-20 token. Access to a competitive USD-denominated return in the

	DSR.
<i>Inefficiency:</i> Acquiring a loan involves costs of time and money.	Instant liquidity at the push of a button with minimal transaction costs.
<i>Lack of Interoperability:</i> Cannot trustlessly use USD or USD-collateralized token in smart contract agreements.	Issuance of DAI, a permissionless USD-tracking stablecoin backed by cryptocurrency. DAI can be used in any smart contract or DeFi application.
<i>Opacity:</i> Unclear collateralization of lending institutions.	Transparent collateralization ratios of vaults visible to entire ecosystem.

6.1.2 Compound

Compound is a lending market that offers several different ERC-20 assets for borrowing and lending. All the tokens in a single market are pooled together so every lender earns the same variable rate and every borrower pays the same variable rate. The concept of a credit rating is irrelevant, and because Ethereum accounts are pseudonymous, enforcing repayment in the event of a loan default is virtually impossible. For this reason, all loans are overcollateralized in a collateral asset different from the one being borrowed. If a borrower falls below their collateralization ratio, their position is liquidated to pay back their debt. The debt can be liquidated by a keeper, similar to the process used in MakerDAO Vaults. The keeper receives a bonus incentive for each unit of debt they close out.

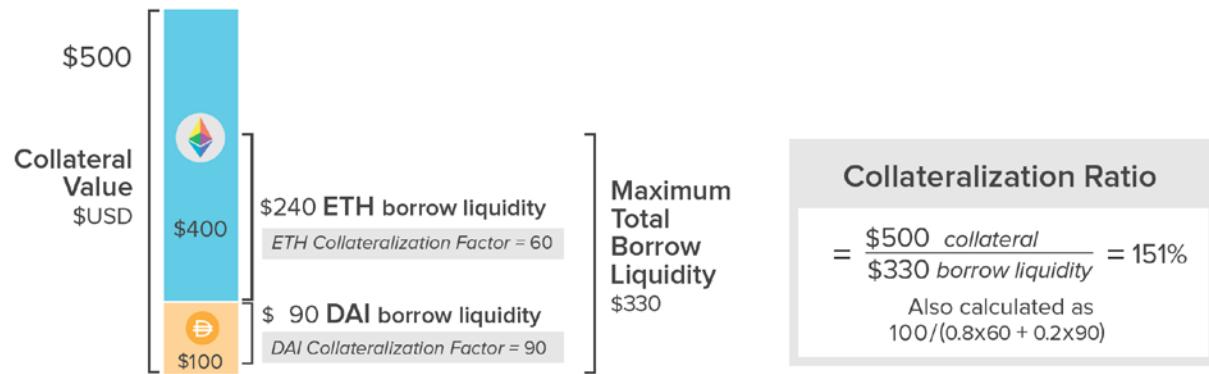
The collateralization ratio is calculated via a *collateral factor*. Each ERC-20 asset on the platform has its own collateral factor ranging from 0-90%. A collateral factor of zero means an asset cannot be used as collateral. The required collateralization ratio for a single collateral type is calculated as 100 divided by the collateral factor. Volatile assets generally have lower collateral factors, which mandate higher collateralization ratios due to increased risk of a price movement that could lead to undercollateralization. An account can use multiple collateral types at once, in which case the collateralization ratio is calculated as 100 divided by the weighted average of the collateral types by their relative sizes (denominated in a common currency) in the portfolio.

The collateralization ratio is similar to a reserve multiplier in traditional banking, constraining the amount of “borrowed” dollars that can be in the system relative to the “real” supply. For instance, there is occasionally more DAI in Compound than is actually supplied by MakerDAO, because users are borrowing and resupplying or selling to others who resupply. Importantly, all MakerDAO supply is ultimately backed by real collateral and there is no way to borrow more collateral value than has been supplied.

For example, suppose an investor deposits 100 DAI with a collateral factor of 90. This transaction alone corresponds to a required collateralization ratio of 111%. Assuming 1 DAI = \$1, the investor can borrow up to \$90 worth of any other asset in Compound. If she borrows the maximum, and the price of the borrowed asset increases at all, the position is subject to liquidation. Suppose she

also deposits two ETH with a collateral factor of 60 and a price of \$200/ETH. The total supply balance is now \$500, with 80% being ETH and 20% being DAI. The required collateralization ratio is $100/(0.8*60 + 0.2*90) = 151\%$.

EXHIBIT B



The supply and borrow interest rates are compounded every block (approximately 15 seconds on Ethereum producing approximately continuous compounding) and are determined by the utilization percentage in the market. Utilization is calculated as total borrow/total supply. The utilization rate is used as an input parameter to a formula that determines the interest rates. The remaining parameters are set by *Compound Governance* which we describe near the end of this section.

The formula for the borrow rate generally is an increasing linear function with a y-intercept known as the *base rate* that represents the borrow rate at 0% borrow demand and a *slope* representing the rate of change of the rates. These parameters are different for each ERC-20 asset supported by the platforms. Some markets have more advanced formulas that include a *kink*. A kink is a utilization ratio beyond which the slope steepens. These formulas can be used to reduce the cost of borrowing up to the kink and then increase the cost of borrowing after the kink to incentivize a minimum level of liquidity.

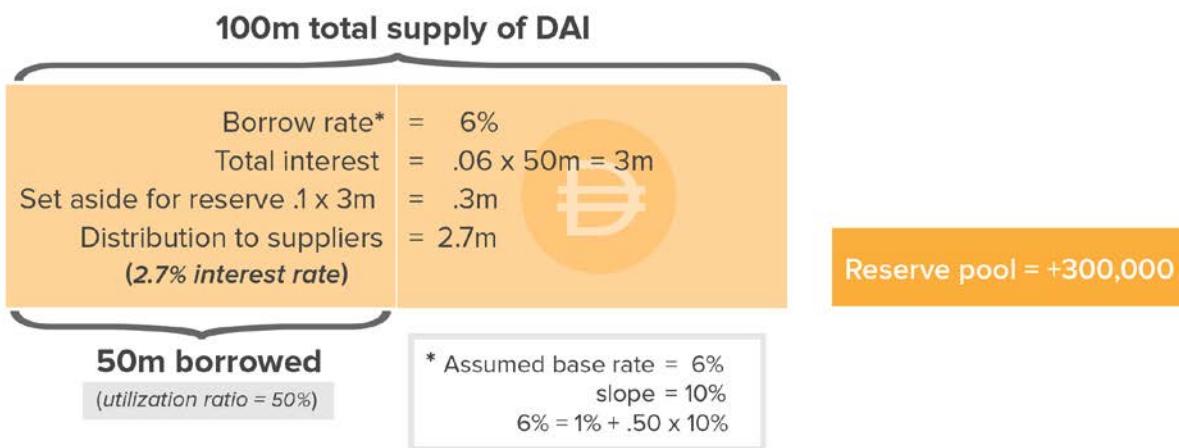
The supply interest rate is the borrow rate multiplied by the utilization ratio so borrow payments can fully cover the supplier rates. The *reserve factor* is a percentage of the borrow payments not given to the suppliers and instead set aside in a reserve pool that acts as insurance in that case a borrower defaults. In an extreme price movement, many positions may become undercollateralized in that they have insufficient funds to repay the suppliers. In the event of such a scenario, the suppliers would be repaid using the assets in the reserve pool.

Here is a concrete example of the rates: In the DAI market, 100 million is supplied and 50 million is borrowed. Suppose the base rate is 1% and the slope is 10%. At 50 million borrowed, utilization is 50%. The borrow interest rate is then calculated to be $0.5*0.1 + 0.01 = 0.06$ or 6%. The maximum supply rate (assuming a reserve factor of zero) would simply be $0.5*0.06 = 0.03$ or 3%. If the reserve factor is set to 10, then 10% of the borrow interest is diverted to a DAI reserve pool,

lowering the supply interest rate to 2.7%. Another way to think about the supply interest rate is that the 6% borrow interest of 50 million is equal to 3 million of borrow payments. Distributing 3 million of payments to 100 million of suppliers implies a 3% interest rate to all suppliers.

For a more complicated example involving a kink, suppose 100 million DAI is supplied and 90 million DAI is borrowed, a 90% utilization. The kink is at 80% utilization, before which the slope is 10% and after which the slope is 40%, which implies the borrow rate will be much higher if the 80% utilization is exceeded. The base rate remains at 1%. The borrow interest rate = 0.01 (base) + 0.8*0.1 (pre-kink) + 0.1*0.4 (post-kink) = 13%. The supply rate (assuming a reserve factor of zero) is 0.9*0.13 = 11.7%.

EXHIBIT C



The utility of the Compound lending market is straightforward: it allows users to unlock the value of an asset without selling it and incurring a taxable event (at least under today's rules), similar to a home equity line of credit. Additionally, they can use the borrowed assets to engineer leveraged long or short positions, with competitive pooled rates and no approval process. For instance, if an investor is bearish on the price of ETH, he can simply deposit a stablecoin, such as DAI or USDC, as collateral, then borrow ETH and sell it for more of the stablecoin. If the price of ETH falls, the investor uses some of the DAI to purchase (cheaply) ETH to repay the debt. Compound offers several volatile and stable tokens to suit the risk preferences of the investor, and new tokens are continually added.

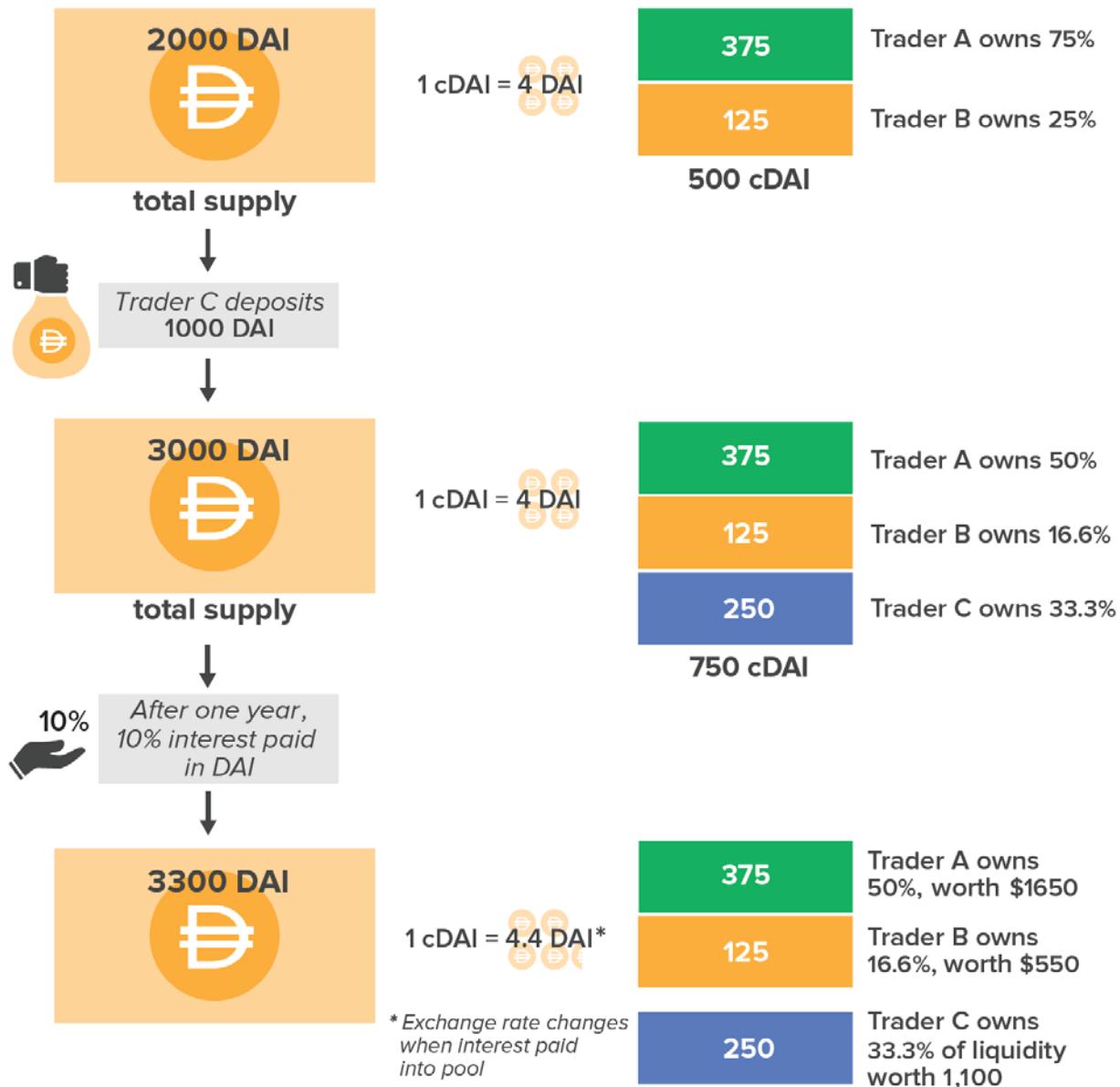
The Compound protocol must escrow tokens as a depositor in order to maintain liquidity for the platform itself and to keep track of each person's ownership stake in each market. A naive approach would be to keep track of the number inside a contract. A better approach would be to tokenize the user's share. Compound does this using a cToken, and this is one of the platform's important innovations.

Compound's cToken is an ERC-20 in its own right that represents an ownership stake in the underlying Compound market. For example, cDAI corresponds to the Compound DAI market and cETH corresponds to the Compound ETH market. Both tokens are minted and burned in

proportion to the funds added and removed from the underlying market as a means to track the amount belonging to a specific investor. Because of the interest payments that continually accrue to suppliers, these tokens are always worth more than the underlying asset. The benefit of designing the protocol in this way is that a cToken can be traded on its own like a normal ERC-20 asset. This trait allows other protocols to seamlessly integrate with Compound simply by holding cTokens and allows users to deploy their cTokens directly into other opportunities, such as using a cToken as collateral for a MakerDAO Vault. Instead of using ETH only as collateral, an investor can use cETH and earn lending interest on the ETH collateral.

For example, assume there are 2,000 DAI in the Compound DAI market and a total 500 cDAI represents the ownership in the market; this ratio of cDAI to DAI is not determinative and could just as easily be 500,000 cDAI. At that moment in time, 1 cDAI is worth 4 DAI, but after more interest accrues in the market the ratio will change. If a trader comes in and deposits 1,000 DAI, the supply increases by 50% (see Exhibit D). Therefore, the Compound protocol mints 50%, or 250, more cDAI and transfers this amount to the trader's account. Assuming an interest rate of 10%, at year end there will be 3,300 DAI, and the trader's 250 cDAI can be redeemed for one-third, or 1,100, of the DAI. The trader can deploy cDAI in the place of DAI so the DAI is not sitting idle but earning interest via the Compound pool. For example, the trader could deploy cDAI as the necessary collateral to open a perpetual futures position on dYdX or she could market make on Uniswap using a cDAI trading pair. (dYdX and Uniswap will be discussed later in the paper.)

EXHIBIT D



The many different parameters of Compound's functionality, such as the *collateral factor*, *reserve factor*, *base rate*, *slope*, and *kink*, can all be tuned. The entity capable of tuning these parameters is *Compound Governance*. Compound Governance has the power to change parameters, add new markets, freeze the ability to initiate new deposits or borrows in a market, and even upgrade some of the contract code itself. Importantly, Compound Governance cannot steal funds or prevent users from withdrawing. In the early stages of Compound's growth, governance was controlled by developer admins, similar to any tech startup. A strong development goal of Compound, as with most DeFi protocols, was to remove developer admin access and release the protocol to the leadership of a DAO via a governance token. The token allowed shareholders and community

members to collectively become Compound Governance and propose upgrades or parameter tuning. A quorum agreement was required for any change to be implemented.²⁰

Compound implemented this new governance system in May 2020 via the COMP token. COMP is used to vote on protocol updates such as parameter tuning, adding new asset support, and functionality upgrades (similar to MKR for MakerDAO). On June 15, 2020, the [7th governance proposal](#) passed which provided for distributing COMP tokens to users of the platform based on the borrow volume per market. The proposal offered an experience akin to a tech company giving its own stock to its users. The COMP token is distributed to both suppliers and borrowers, and acts as a subsidization of rates. With the release of the token on public markets, COMP's market cap spiked to over \$2 billion. The price point of the distribution rate is so high that borrowing in most markets turned out to be profitable. This arbitrage opportunity attracted considerable volume to the platform, and the community governance has made and passed several proposals to help manage the usage.

The Compound protocol can no longer be turned off and will exist on Ethereum as long as Ethereum exists. Other platforms can easily escrow funds in Compound to provide additional value to their users or enable novel business models. An interesting example of this is [PoolTogether](#). PoolTogether is a no-loss lottery²¹ that deposits all user's funds into Compound, but pays the entire pool's earned interest to a single random depositor at fixed intervals. Easy, instant access to yield or borrow liquidity on different Ethereum tokens makes Compound an important platform in DeFi.

Traditional Finance Problem	Compound Solution
<i>Centralized Control:</i> Borrowing and lending rates are controlled by institutions.	Compound rates are determined algorithmically and gives control of market parameters to COMP stakeholders incentivized to provide value to users.
<i>Limited Access:</i> Difficulty in accessing high-yield USD investment opportunities or competitive borrowing.	Open ability to borrow or lend any supported assets at competitive algorithmically determined rates (temporarily subsidized by COMP distribution).
<i>Inefficiency:</i> Suboptimal rates for borrowing and lending due to inflated costs.	Algorithmically pooled and optimized interest rates.
<i>Lack of Interoperability:</i> Cannot repurpose supplied positions for other investment opportunities.	Tokenized positions via cTokens can be used to turn static assets into yield-generating assets.

²⁰ The quorum rule for Compound is a majority of user each of whom holds with a minimum of 400,000 COMP (~4% of total eventual supply)

²¹ In most lotteries, 30-50% of the lottery sales are tagged for administrative costs and government or charitable use; hence, the expected value of investing \$1.00 in a lottery is \$0.50-\$0.70. In a no-loss lottery, all sales are paid out and the expected value is \$1.00.

<i>Opacity:</i> Unclear collateralization of lending institutions.	Transparent collateralization ratios of borrowers visible to entire ecosystem.
--	--

6.1.3 Aave

[Aave](#) (launched in 2017) is a lending market protocol similar to Compound and offers several enhanced features. Aave offers many additional tokens to supply and borrow beyond what Compound offers. At the time of writing, Compound offers eight distinct tokens (different ERC-20 Ethereum-based assets) and Aave offers these eight plus an additional nine not offered on Compound. Importantly, the Aave lending and variable borrowing rates are more predictable, because unlike the volatile COMP token in Compound, no subsidy is involved.

The Aave protocol supports the ability to create entirely new markets. Each market consists of its own group of token pools with their corresponding supply and borrow interest rates. The benefit of creating a separate market is that the market's supported tokens act as collateral solely in that market and cannot affect other markets, thus mitigating any potential contagion.

Aave currently has two main markets. The first is for more-conventional ERC-20 tokens similar to those of Compound, supporting assets such as ETH, USDC, and DAI. The second is specific to Uniswap LP tokens. For example, when a user deposits collateral into a Uniswap market, she receives an LP token that represents her ownership in the market. The LP tokens can be deposited in the Uniswap market on Aave to generate additional returns.

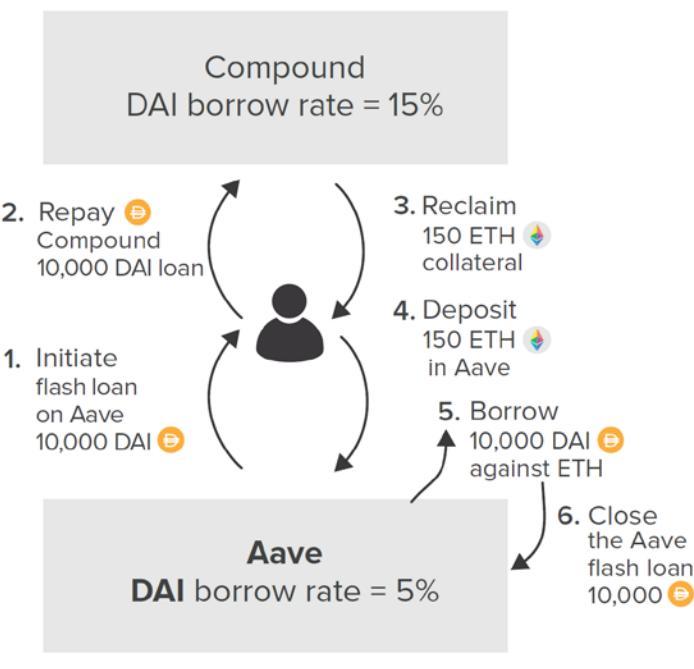
Aave also supports flash loans (discussed in section 5) in all of its markets and is the only source of flash liquidity for many smaller-cap tokens. Aave charges a fee of 9 basis points (bps) on the loan amount to execute a flash loan. The fee is paid to the asset pool and provides an additional return on investment to suppliers, because they each own a pro rata share of the pool. An important use case for flash loans is that they allow users quick access to capital as a means to refinance positions. This functionality is crucial to DeFi, both as general infrastructure and as a component of a positive user experience (UX).

To provide an example, assume the price of ETH is 200 DAI. A user supplies 100 ETH in Compound and borrows 10,000 DAI to lever up and purchase an additional 50 ETH, which the user also supplies to Compound. Suppose the borrow interest rate in DAI on Compound is 15% but only Aave is 5%. The goal is to refinance the borrowing to take advantage of the lower rate offered on Aave, which is analogous to refinancing a mortgage, a long and costly process in centralized finance.

One option is to manually unwind each trade on Compound and re-do both trades on Aave to reconstruct the levered position, but this option is wasteful in terms of exchange fees and gas fees. The easier action is to take out a flash loan from Aave for 10,000 DAI, use it to pay the debt on Compound, withdraw the full 150 ETH, resupply to Aave, and trigger a normal Aave borrow position (at 5% APR) against that collateral to repay the flash loan. The latter approach effectively skips the steps of exchanging ETH for DAI to unwind and rewind the leverage.

EXHIBIT G

Before + 150 ETH (collateral)  – 10,000 DAI (loan)  at 15% interest



After + 150 ETH (collateral)  – 10,000 DAI (loan)  at 5% interest

As shown in the preceding example, a flash loan used to refinance a position allows for DeFi client applications that let users migrate a levered position from one dApp to another with the single push of a button. These applications can even optimize portfolios for APR among several competing offerings including Maker DSR (Dai Savings Rate), Compound, dYdX, and Aave.

An Aave innovation (and as of this writing only available on Aave) is a “stable” rate loan. The choice of “stable” intentionally avoids the use of “fixed rate.” A borrower has the option to switch between the variable rate and the current stable rate. The supply rate is always variable, because under certain circumstances, such as if all borrowers left the market, it would be impossible to fund a fixed supply rate. The suppliers always collectively earn the sum of the stable and variable borrow interest payments minus any fees to the platform.

The stable rate is not a fixed rate, because the rate is adjustable in extreme liquidity crunches and can be refinanced to a lower rate if market conditions allow. Also, some constraints exist around how much liquidity can be removed at a specific stable rate. Algorithmic stable borrowing rates provide value to risk-averse investors who wish to take on leverage without the uncertainty of a variable-rate position.

Aave is developing a *Credit Delegation* feature in which users can allocate collateral to potential borrowers who can use it to borrow a desired asset. The process is unsecured and relies on trust. This process allows for uncollateralized loan relationships, such as in traditional finance, and potentially opens the floodgates in terms of sourcing liquidity. The credit delegation agreements will likely have fees and credit scores to compensate for the risk of unsecured loans. Ultimately, the delegator has sole discretion to determine who is an eligible borrower and what contract terms are sufficient. Importantly, credit delegation terms can be mediated by a smart contract. Alternatively, the delegated liquidity can be given to a smart contract, and the smart contract can use the liquidity to accomplish its intended function. The underlying benefit of credit delegation is that all loans in Aave are ultimately backed by collateral, regardless of whose collateral it is.

For example, a supplier may have a balance of 40,000 DAI in Aave earning interest. The supplier wants to increase their expected return via an unsecured delegation of their collateral to a trusted counterparty. The supplier likely knows the counterparty through an off-chain relationship, perhaps it is a banking client. The counterparty can proceed to borrow, for instance, 100 ETH with the commitment to repay the asset to the supplier plus an agreed-upon interest payment. The practical impact is that the external relationship is unsecured because no collateral is available to enforce payment; the relationship is based essentially on trust.

In summary, Aave offers several innovations beyond the lending products offered by Compound and other competitors. Aave's flash loans, although not unique among competitors, provide additional yield to investors, making them a compelling mechanism to provide liquidity. These utilities also attract to the platform arbitrageurs and other applications that require flash liquidity for their use cases. Stable borrow rates are a key innovation, and Aave is the only platform currently with this offering. This feature could be important for larger players who cannot operate under the potential volatility of variable borrow rates.

Finally, *Credit Delegation* allows users to unlock the value of supplied collateral in novel ways, including through traditional markets and contracts and even via additional layers of smart contracts that charge a premium rate to compensate for risk. Credit delegation allows loan providers to take their own collateral in the form of nonfungible Ethereum assets, perhaps tokenized art or real estate not supported by the main Aave protocol. As Aave continues to innovate, the platform will continue to amass more liquidity and cover a wider base of potential use cases.

Traditional Finance Problem	Aave Solution
<i>Centralized Control:</i> Borrowing and lending rates controlled by institutions.	Aave interest rates are controlled algorithmically.
<i>Limited Access:</i> Only select groups have access to large quantities of money for arbitrage or refinancing.	Flash loans democratize access to liquidity for immediately profitable enterprises.
<i>Inefficiency:</i> Suboptimal rates for borrowing	Algorithmically pooled and optimized interest

and lending due to inflated costs.	rates.
<i>Lack of Interoperability:</i> Cannot monetize or utilize excess collateral in a lending position.	Credit delegation allows parties to use deposited collateral when they do not need borrowing liquidity.
<i>Opacity:</i> Unclear collateralization of lending institutions.	Transparent collateralization ratios of borrowers visible to the entire ecosystem.

6.2 Decentralized Exchange

6.2.1 Uniswap

The primary example of an AMM on Ethereum is [Uniswap](#). Currently, Uniswap uses a constant product rule to determine the trading price, using the formula $k = x^*y$, where x is the balance of asset A, and y the balance of asset B. The product k is the *invariant* and is required to remain fixed at a given level of liquidity. To purchase (withdraw) some x , some y must be sold (deposited). The implied price is x/y and is the *risk-neutral* price, because the contract is equally willing to buy or sell at this rate as long as invariant k is constant.

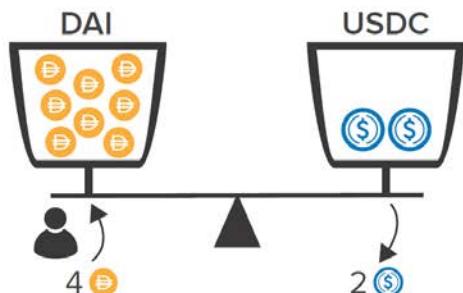
Consider a concrete example. For simplicity, we will ignore transaction fees (gas) in all of the examples. Assume an investor in the Uniswap USDC/DAI market has 4 DAI (Asset A) and 4 USDC (Asset B). This sets the instantaneous exchange rate at 1 DAI:1 USDC and the invariant at 16 ($= x^*y$). To sell 4 DAI for USDC, the investor deposits 4 DAI to the contract and withdraws 2 USDC. Now the USDC balance is $4 - 2 = 2$ and the DAI balance is $4 + 4 = 8$. The invariant remains constant at 16. Notice that the effective exchange rate was 2 DAI: 1 USDC. The change in the exchange rate is due to slippage because of the low level of liquidity in the market. The magnitude of the invariant determines the amount of slippage. To extend the example, assume the balance is 100 DAI and 100 USDC in the contract. Now the invariant is 10,000, but the exchange rate is the same. If the investor sells 4 DAI for USDC, now 3.85 USDC can be withdrawn to keep the invariant constant and results in much lower slippage at an effective rate of 1.04 DAI: 1 USDC.

EXHIBIT E



Scenario A

Exchange 4 DAI



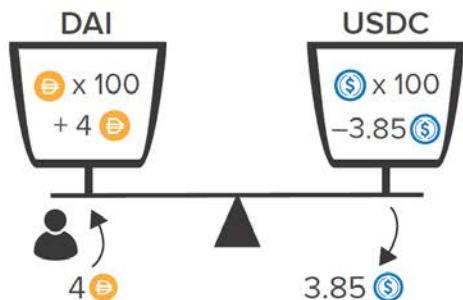
$$\text{Invariant} = K = 8 \text{ DAI} \times 2 \text{ USDC} = 16$$

Hence, 4 DAI exchanged for 2 USDC

Scenario B

Exchange 4 DAI

but contract has more liquidity, 100 DAI, 100 USDC



$$\text{Instantaneous exchange rate} = 1 \text{ DAI} = 1 \text{ USDC}$$

$$\text{Before } K = 100 \times 100 = 10,000$$

$$\text{After } K = 104 \times 96.15 = 10,000$$

$$\text{Implied price} = 1.04 \text{ DAI} = 1 \text{ USDC}$$

Deep liquidity helps minimize slippage. Therefore, it is important that Uniswap incentivizes depositors to supply capital to a given market. Anyone can become a liquidity provider by supplying assets on both sides of a market at the current exchange rate.²² Supplying both sides increases the product of the amount of assets held in the trading pair (i.e., increases the invariant as mentioned in the formula for the market maker). Per the preceding example, higher invariants lead to lower slippage and therefore an increase in effective liquidity. We can think of the invariant as a direct measure of liquidity. In summary, liquidity providing increases the invariant with no effect on price, whereas trading against a market impacts the price with no effect on the invariant.

Each trade in a Uniswap market has an associated 0.3% fee that is paid back into the pool. Liquidity providers earn these fees based on their pro rata contribution to the liquidity pool. They therefore prefer high-volume markets. This mechanism of earning fees is identical to the *cToken*

²² A liquidity provider adds to both sides of the market, thereby increasing total market liquidity. If a user exchanges one asset for another, the total liquidity of the market as measured by the invariant does not change.

model of Compound. The ownership stake is represented by a similar token called a *Uni* token. For example, the token representing ownership in the DAI/ETH pool is Uni DAI/ETH.

Liquidity providers in Uniswap essentially earn passive income in proportion to the volume on the market they are supplying. Upon withdrawal, however, the exchange rate of the underlying assets will almost certainly have changed. This shift creates an opportunity-cost dynamic (*impermanent loss*) that arises because the liquidity provider could simply hold the underlying assets and profit from the price movement. The fees earned from trading volume must exceed impermanent loss in order for liquidity providing to be profitable. Consequently, stablecoin trading pairs such as USDC/DAI are attractive for liquidity providers because the high correlation of the assets minimizes the impermanent loss.

Uniswap's $k = x^*y$ pricing model works well if the correlation of the underlying assets is unknown. The model calculates the exact same slippage at a given liquidity level for any two trading pairs. In practice, however, we would expect much lower slippage for a stablecoin trading pair than for an ETH trading pair, because we know by design that the stablecoin's price should be close to \$1. The Uniswap pricing model leaves money on the table for arbitrageurs on high correlation pairs such as stablecoins, because it does not adjust default slippage lower (change the shape of the bonding curve), as would be expected; the profit is subtracted from the liquidity providers. For this reason, competitor AMMs, such as [Curve](#), that specialize in high-correlation trading pairs may cannibalize liquidity in these types of Uniswap markets.

Anyone can start an ERC-20/ERC-20 or ETH/ERC-20 trading pair on Uniswap, if the pair does not already exist, by simply supplying capital on both sides.²³ The user determines the initial exchange rate, and arbitrageurs should drive that price to the true market price if it deviates at all. Users of the platform can effectively trade any two ERC-20 tokens supported by using *router contracts* that determine the most efficient path of swaps in order to get the lowest slippage, if no direct trading pair is available.

A drawback of the AMM model is that it is particularly susceptible to “front-running”. This is not to be confused with illegal front-running which plagues centralized finance. One of the features of blockchain is that all transactions are public. That is, when an Ethereum user posts a transaction to the memory pool, it is publicly visible to all Ethereum nodes. Front-runners can see this transaction which is public information and post a higher gas-fee transaction to trade against the pair before the user’s transaction is added to a block, and then immediately trade in the reverse direction against the pair. This strategy allows a user to easily profit from large transactions, especially in illiquid markets with high slippage. For this reason, Uniswap allows users to set a maximum slippage as a clause in the transaction. If the acceptable level of slippage is exceeded,

²³ ETH, although fungible, is not an ERC-20. Many platforms, including Uniswap, instead use [WETH](#), an ERC-20-wrapped version of ETH to get around this. Uniswap allows a user to directly supply and trade with ETH and it converts to WETH behind the scenes.

the trade will fail to execute.²⁴ This provides a limit to the profit front-runners can make, but does not completely remove the problem.

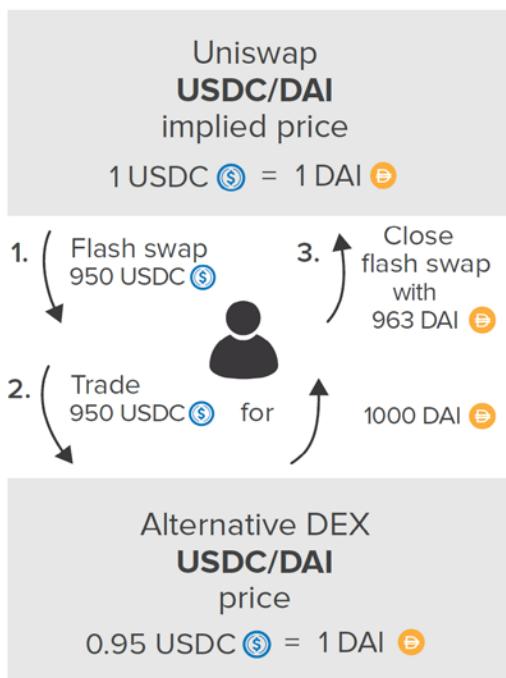
Another drawback is that arbitrage profits go only to arbitrageurs, who do not have a vested interest in the platform. The arbitrageurs profit at the expense of liquidity providers, who should not be losing the potential spread they would earn in a normal market-making scenario. Competing platforms, such as [Mooniswap](#), propose to solve this issue by supplying virtual prices that slowly approach the true price, leaving tighter time windows and lower spreads for arbitrageurs to capitalize on. The additional spread remains in the pool for the liquidity providers.

Uniswap offers an interesting feature, a *flash swap*, similar to a flash loan, (described in Section 5) called *flash swaps*. In a flash swap, the contract sends the tokens *before* the user pays for them with assets on the other side of the pair. A flash swap unlocks many opportunities for arbitrageurs. The user can deploy this instant liquidity to acquire the other asset at a discount on another exchange before repaying it; the corresponding amount of the alternate asset must be repaid in order to maintain the invariant. This flexibility in a flash swap is different from the provision in a flash loan, which requires that repayment occur with the same asset. A key aspect of a flash swap is that all trades must take place during a single Ethereum transaction and that the trade must be closed with the corresponding amount of the complementary asset in that market.

Consider this example in the DAI/USDC market with a supply of 100,000 each. This implies a 1:1 exchange rate and an invariant of 10 billion. A trader who has no starting capital spots an arbitrage opportunity to buy DAI on a DEX for 0.95 USDC. The trader can capitalize on this arbitrage via a flash swap by withdrawing 950 USDC of flash liquidity (liquidity derived from a flash loan) from the DAI/USDC market, purchase 1,000 DAI via the described arbitrage trade, and repay 963 DAI for a profit of 37 DAI—all consummated with no initial capital. The figure of 963 is calculated as 960 (with rounding for ease of illustration) to maintain the 10 billion invariant, and to account for some slippage, plus a $0.30\% \times 960 = 3$ DAI transaction fee paid into the pool owned by the liquidity providers.

²⁴This is a smart contract level check. In other words, before finalizing the trade, the contract checks the total slippage from the initially posted price to the effective execution price (which could have changed if other transactions made it in first like the described front running attempt). If this slippage exceeds the pre-defined user tolerance, the entire trade is cancelled and the contract execution fails.

EXHIBIT F



4. Slippage = 10 DAI, so 960 DAI
- Fee = $.003 \times 960 = 3 \text{ DAI}$
- Swap done at $960 + 3 = 963 \text{ DAI}$
- Profit = $1000 - 963 = 37 \text{ DAI}$

Lastly, an important point about Uniswap is the release of a governance token in September 2020 called UNI. Like COMP, the Compound governance token, UNI is distributed to users to incentivize liquidity in key pools including ETH/USDC and ETH/DAI. The UNI governance even has some control over its own token distribution because 43% of the supply will be vested over four years to a treasury controlled by UNI governance. Importantly, each unique Ethereum address that had used Uniswap before a certain cutoff date (over 250,000 addresses) was given 400 UNI tokens as a free airdrop. At the same time as the airdrop, UNI was released on Uniswap and the Coinbase Pro exchange for trading. The price per token opened around \$3 with a total market cap of over \$500 million, amounting to \$1,200 of liquid value distributed directly to each user. This flood of supply could have led to selling pressure that tanked the token price. Instead, the token price spiked to over \$8 before settling in the \$4–5 range. Through UNI, Uniswap effectively crowdsourced capital to build and scale its business, which attained a unicorn valuation for a short time. This demonstrates the value the community places in the token and the platform, because the majority of supply is still held by those who received the airdrop.

As evidence that Uniswap is a good idea, it has been largely copied by [Sushiswap](#). Furthermore, the CFMM has been generalized by [Balancer](#). With Balancer, more than two markets can be

supported in a liquidity pool. In addition, the assets can be arbitrarily weighted (currently, Uniswap requires equal value).²⁵ Further, the liquidity pool creator sets the transaction fees.

As of March 2021, the Uniswap team released a timeline and upgrade plan for the Uniswap protocol. Termed Uniswap v3, the Uniswap team proposed several changes to the protocol's liquidity provisioning model, moving away from the constant product formula described earlier and towards a model that resembles an on-chain, limit order book.²⁶ This change increases Uniswap's flexibility, allowing users and liquidity providers to customize curves and more actively manage their liquidity positions/control their return profiles.

Uniswap is critical infrastructure for DeFi applications; it is important to have exchanges operational whenever it is needed. Uniswap offers a unique approach for generating yield on users' assets by being a liquidity provider. The platform's flash swap functionality aids arbitrageurs in maintaining efficient markets and unlocks new use cases for users. Users can access any ERC-20 token listed, including creating completely new tokens through an IDO. As AMM volume grows on Ethereum and new platforms arise with competing models, Uniswap will continue to be a leader and an example of critical infrastructure going forward.

Traditional Finance Problem	Uniswap Solution
<i>Centralized Control:</i> Exchanges that control which trading pairs are supported.	Allows anyone to create a new trading pair if it does not already exist and automatically routes trades through the most efficient path if no direct pair exists.
<i>Limited Access:</i> The best investment opportunities and returns from liquidity providing are restricted to large institutions.	Anyone can become a liquidity provider and earn fees for doing so. Any project can list its token on Uniswap to give anyone access to an investor.
<i>Inefficiency:</i> Trades generally require two parties to clear.	An AMM that allows constant access for trading against the contract.
<i>Lack of Interoperability:</i> Ability to exchange assets on one exchange is not easily used within another financial application.	Any token swap needed for a DeFi application can utilize Uniswap as an embedded feature.
<i>Opacity:</i> Unknown if the exchange truly owns all user's entire balance.	Transparent liquidity levels in the platform and algorithmic pricing.

²⁵ The bonding surface in Balancer is given by $V = \prod_{t=0}^n B_t^{W_t}$ where V is the value function (analogous to k), n is the number of assets in the pool, B is the balance of the token t in the pool and W is the normalized weight of token t . See: <https://medium.com/balancer-protocol/bonding-surfaces-balancer-protocol-ff6d3d05d577>

²⁶ <https://uniswap.org/blog/uniswap-v3/>

6.3 Derivatives

6.3.1 Yield Protocol

[Yield Protocol](#) proposes a derivative model for secured, zero-coupon bonds. Essentially, the protocol defines a *yToken* to be an ERC-20 (fungible) token that settles in some fixed quantity of a target asset at a specified date. The contract will specify that the tokens, which have the same expiry, target asset, collateral asset, and collateralization ratio, are fungible. The tokens are secured by the collateral asset and have a required maintenance collateralization ratio similar to, for example, MakerDAO, as well as to other DeFi platforms we have discussed. If the collateral's value dips below the maintenance requirement, the position can be liquidated with some or all of the collateral sold to cover the debt.

The mechanism for *yToken* settlement is still undecided, but one proposed solution is “cash” settlement, which means paying an equivalent amount of the collateral asset worth the specified amount of the target asset. For example, if the target asset is 1 ETH secured by 300 DAI, and at expiry $1 \text{ ETH} = 200 \text{ DAI}$, a cash settlement would pay out 200 DAI and return the 100 DAI excess collateral to the seller of the *yToken*.

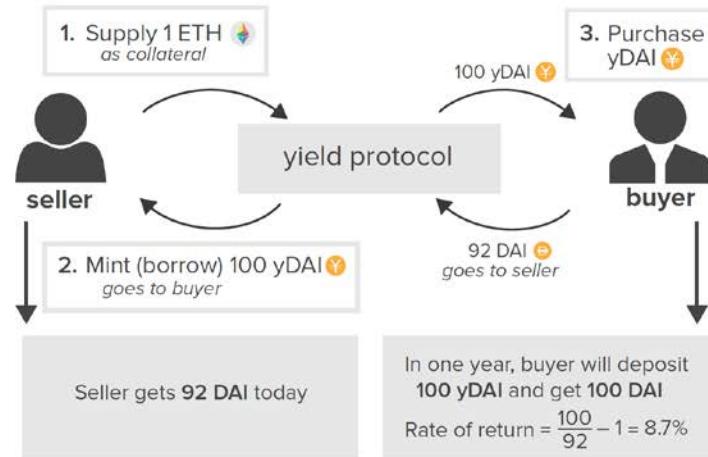
The other commonly proposed solution is “physical” settlement, which automatically sells collateral for the target asset upon expiry (perhaps on Uniswap) to pay out in the target asset. Using the same numbers as in the previous example, the owner of the *yToken* would receive 1 ETH and the seller would receive slightly less of the remaining collateral, likely around 95 DAI, after subtracting exchange fees.

The *yToken* effectively allows for fixed-rate borrowing and lending, using the implied return on the discounted price of the token versus the target amount.

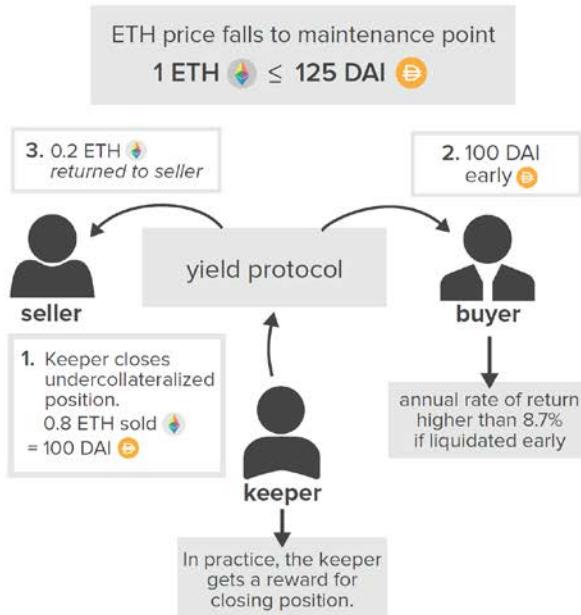
We can illustrate as follows: assume a user has a *yToken* with the target asset of 1 DAI backed by ETH. The maturity date is one year ahead and the *yToken* is trading at 0.92 DAI. A purchase of the *yToken* effectively secures an 8.7% fixed interest rate, even in the case of a liquidation. In the event of a normal liquidation, the collateral would be sold to cover the position, as shown in Exhibit H.

EXHIBIT H

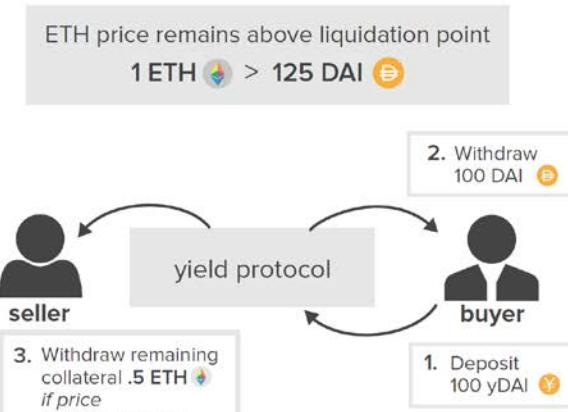
1 ETH 🍁 = 200 DAI 💸
collateralization ratio: **125%**



Scenario A



Scenario B



A compelling third option for settlement (in addition to cash and physical) is “synthetic” settlement. Here, the underlying asset is not directly repaid, but instead rolled into an equivalent amount of that asset pool on a lending platform such as Compound. Synthetic settlement means that yDAI could settle in cDAI, converting the fixed rate into a floating rate. The buyer could close

the position and redeem cDAI for DAI at her leisure. The Yield Protocol handles all of these conversions for the user so that UX simply revolves around the target asset.

In the [Yield Protocol white paper](#), the authors discuss interesting applications from the investor's perspective. An investor can purchase yTokens to synthetically lend the target asset. The investor would be paying X amount of the asset now to purchase the yTokens. Upon settlement, the investor receives X + interest. This financial transaction in total is functionally a lend of the target asset. Note that the interest is implied in the pricing and not a directly specified value. Alternatively one can mint and sell yTokens to synthetically borrow the target asset. By selling a yToken, you are receiving X amount of the asset now (the face value) and promising to pay X + interest in the future. This financial transaction is functionally a borrow of the target asset.

Additional applications include a perpetual product on top of yTokens that maintains a portfolio of different maturities and rolls short-term profits into long term yToken contracts. For example, the portfolio may include 3-month, 6-month, 9-month, and 1-year maturity yTokens, and once the 3-month tokens mature the smart contract can reinvest the balance into 1-year maturity yTokens. Token holders in this fund would essentially be experiencing a floating rate yield on the underlying asset with rate updates every three months.

The yTokens also allow for the construction of yield curves by analyzing the implied yields of short and longer term contracts. This can allow observers to get insights into investor sentiment among the various supported target assets.

The Yield Protocol can even be directly used to speculate on interest rates. There exist a few DAI derivative assets that represent a variable interest rate (Compound cDAI, Aave aDAI, [Chai](#)). One can imagine a seller of yDAI using one of these DAI derivative assets as collateral. The effect of this transaction is that the seller is paying the fixed rate on the yDAI while receiving the variable rate on the collateral. This is a bet that rates will increase. Likewise purchasing yDAI (of any collateral type) is a bet that variable rates will NOT increase beyond the fixed rate received.

Yield is an important protocol that supplies fixed rate products to Ethereum. It can be tightly integrated with other protocols like MakerDAO and Compound to create robust interest-bearing applications for investors. Demand for fixed income components will grow as mainstream investors begin adopting DeFi with portfolios in need of these types of assets.

Traditional Finance Problem	Yield Solution
<i>Centralized Control:</i> Fixed income instruments largely restricted to governments and large corporations.	Yield protocol is open to parties of any size.
<i>Limited Access:</i> Many investors have limited access to buy or sell sophisticated fixed income investments.	Yield allows any market participant to buy or sell a fixed income asset that settles in a target asset of their choosing.

<i>Inefficiency:</i> Fixed income rates are lower due to layers of fat in traditional finance.	Lean infrastructure running on Ethereum allows for more competitive rates and diverse liquidity pools due to the elimination of middlemen.
<i>Lack of Interoperability:</i> Fixed income instruments generally settle in cash which the investor must determine how to allocate.	yTokens can settle in any Ethereum target asset and even settle synthetically into a floating-rate lending protocol to preserve returns.
<i>Opacity:</i> Risk and uncertainty of counterparty in traditional agreements.	Clear collateralization publicly known on Ethereum blockchain backing the investment.

6.3.2 dYdX

[dYdX](#) is a company that specializes in margin trading and derivatives. The margin trading protocol supports USDC, DAI, and ETH. The company has a spot DEX that allows investors to exchange these assets against the current bid–ask on the order book. The DEX uses a hybrid on–off chain approach. Essentially dYdX stores *signed* or pre-approved orders without submitting to Ethereum. These orders use cryptography to guarantee they are only used to exchange funds for the desired asset at the desired price. The DEX supports limit orders and a *maximum slippage* parameter for market orders in an effort to mitigate the slippage associated with price moves or front running.

dYdX provides market makers and traders the open-source software required to interact with the DEX; alternatively, users can simply use the user interface (UI). Having dYdX do the order matching introduces a certain element of trust, because the infrastructure could be in downtime or not posting transactions for some reason. Allowing dYdX to match the orders holds little or no risk that the company could steal user funds, because the signed orders can only be used as intended per the smart contract. When the orders are matched, they are submitted to the Ethereum blockchain, where the smart contract facilitates settlement.

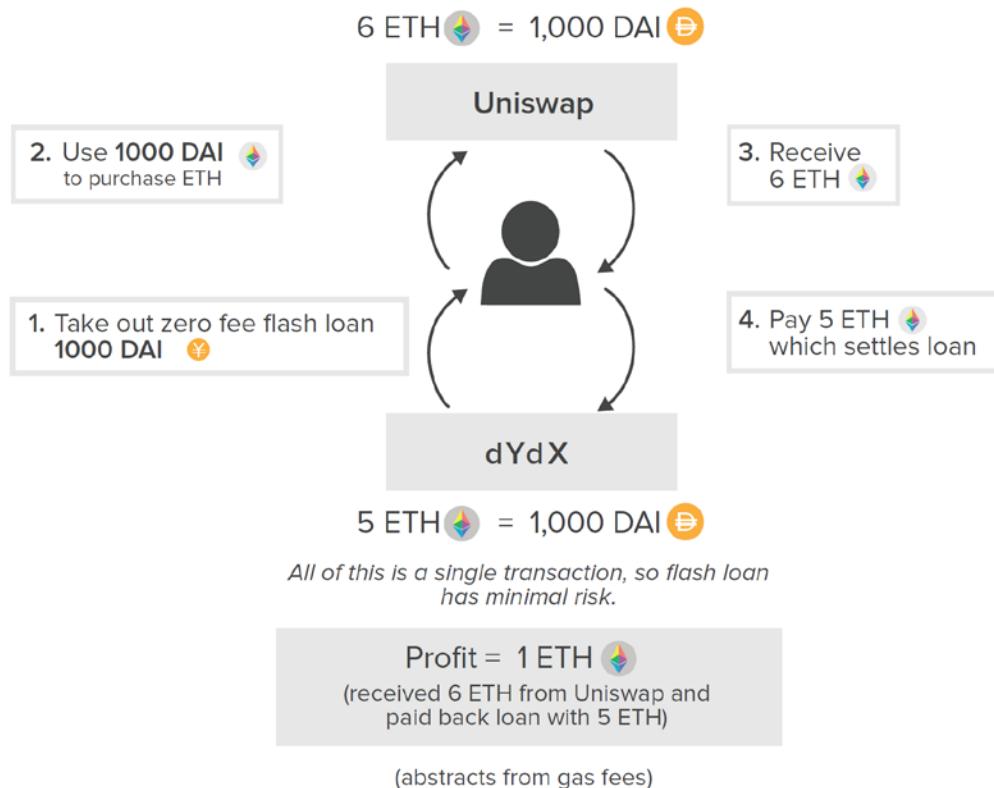
In addition, an investor can take a levered long or short position using margined collateral. The maximum leverage dYdX allows is 10 times. The positions can be isolated so that a single collateral deposit is used or cross-margined so that all of the investor's balances are pooled to serve as collateral. As in other protocols, dYdX has a maintenance margin requirement that if not maintained triggers liquidation of the collateral to close the position. The liquidations can be performed by external keepers who are paid to find and liquidate underwater positions, similar to the process followed by MakerDAO.

dYdX offers borrowing and lending similar to Compound and Aave. The dYdX markets also feature flash loans. Unlike Aave, the flash loans are free, so that dYdX is a popular choice for DAI, ETH, and USDC flash liquidity. In the world of open smart contracts, it makes sense that flash loans rates would be driven to zero given that they are near risk free. Lending rates are determined by the loan's duration and relative risk of default. For flash loans, repayment is algorithmically

enforced and time is infinitesimal: in a single transaction, only the user can make any function calls or transfers. No other Ethereum users can move funds or make any changes while a particular user's transaction is in flight, resulting in no opportunity cost for the capital. Hence, as expected, a market participant offering free flash loans will attract more usage to their platform. Because flash loans do not require any upfront capital, they democratize access to funds for various use cases. In the Aave example, we showed how flash loans can be used to refinance a loan. We will now illustrate the use of flash loans to capitalize on an arbitrage opportunity.

Suppose the effective exchange rate for 1,000 DAI for ETH on Uniswap is 6 ETH/1,000 DAI. (The instantaneous exchange rate would be different, due to slippage.) Also, suppose the dYdX DEX has a spot ask price of 5 ETH for 1,000 DAI (i.e., the ETH are much more expensive on dYdX than Uniswap). To capitalize on this arbitrage opportunity, without any capital beyond the gas fee, an investor can execute a flash loan to borrow 1,000 DAI, exchange it on Uniswap for 6 ETH, and use 5 of those ETH to trade for 1,000 DAI on dYdX. Finally, the investor can repay the flash loan with the 1,000 DAI and pocket the 1 ETH profit. This all happens in a single transaction; multiple contract executions can happen in a single transaction on the Ethereum blockchain. See Exhibit I.

EXHIBIT I



The main derivative product dYdX offers is a BTC perpetual futures contract. Perpetual futures are a popular derivative product similar to traditional futures but without a settlement date. By entering into a perpetual futures contract, the investor is simply betting on the future price of an asset. The contract can be long or short, with or without leverage. The perpetual futures contract

uses an Index Price based on the average price of the underlying asset across the major exchanges.²⁷ The investor deposits margin collateral and chooses a direction and amount of leverage. The contract can trade at a premium or discount to the Index Price depending on whether more traders are long or short the underlying, in this case BTC.

A funding rate, paid from one side to the other, keeps the futures price close to the Index. If the futures contract is trading at a premium to the Index, the funding rate would be positive, and longs would pay shorts. The magnitude of the funding rate is a function of the difference in price compared to the Index. Likewise, if the contract is trading at a discount, the shorts pay the long positions. The funding rate incentivizes investors to take up the opposing side from the majority in order to keep the contract price close to the Index.²⁸ As long as the required margin is maintained, the investor can always close the position at the difference in the price of the notional position minus any negative balance held on margin.

Like a traditional futures contract, the perpetual futures contract has two margins: initial and maintenance. Suppose the initial margin is 10%. This means the investor needs to post collateral (or equity) worth 10% of the underlying asset. A long futures contract allows the investor to buy the asset at a set price in the future. If the market price rises, the investor can buy the asset at a price cheaper than the market price and the profit is the difference between the market price and the contract price. A short position works similarly except that the investor agrees to sell the asset at a fixed price. If the market price falls, the investor can purchase the asset in the open market and sell at the higher price stipulated in the contract. The profit is the difference between the contract price and the market price.

The risk is that the price moves against the investor. For example, if the investor is long with a 10% margin and the market price drops by 10%, the collateral is gone because the difference between purchasing at the contract price and selling in the open market (at a loss) wipes out the value of the collateral. Importantly, futures are different from options. If the underlying asset's price moves the wrong way in an option contract, the option holder can walk away. The exercise of the option is discretionary—that's why it is called an option—and no trader would exercise an option to guarantee a loss. Futures, however, are obligations. As such, traditional exchanges have mechanisms that seek to minimize the chance the contract holder does not default on a losing position.

The maintenance margin is the main tool to minimize default. Suppose the maintenance margin is 5%. On a traditional futures exchange, if the price drops by 5% the investor is required to

²⁷ BTC-USD Perpetual uses the MakerDAO BTCUSD Oracle V2, an oracle that reports in on-chain fashion the bitcoin prices from the cryptocurrency exchanges of Binance, Bitfinex, Bitstamp, Bittrex, Coinbase Pro, Gemini, and Kraken. See <https://defiprime.com/perpetual-dydx>

²⁸ Each protocol in DeFi can only update balances when a user interacts with the protocol. In the example of Compound, the interest rate is fixed until supply enters or leaves the pool which changes the utilization. The contract simply keeps track of the current rate and the last timestamp when the balances updated. When a new user borrows or supplies, that transaction updates the rates for the entire market. Similarly, whereas the dYdX's Funding Rate is updated every second, it is only applied at the time a user opens, closes, or edits a position. The contract calculates the new values based on what the rates were and how long the futures position has been open.

replenish the collateral to bring it back up to 10%. If the investor fails to do this, the exchange liquidates the position. A similar mechanism exists on dYdX, but with important differences. First, if any position falls to 5%, keepers will trigger liquidation. If any collateral remains, they may keep it as a reward. Second, the liquidation is almost instantaneous. Third, no centralized exchange exists. Fourth, dYdX contracts are perpetual, whereas traditional exchange contracts usually have a fixed maturity date.

Consider the following example. Suppose the BTC price index is 10,000 USDC/BTC. An investor initiates a long position by depositing 1,000 USDC as margin (collateral), creating a levered bet on the price of BTC. If the price rises by 5%, the profit is 500. Given the investor has only deposited 1,000, the investor's rate of return is 50%, or $(1,000 - 500)/1,000$.

We can also think about the mechanics another way. Taking a long position at 10,000, the investor is committing to buying at 10,000 and the obligation is 10,000. Think of the obligation as a "negative balance" because the investor must pay 10,000 according to the contract. The investor has already committed collateral of 1,000 and owes 9,000. On the other side, the investor has committed those funds to purchase an asset, 1 BTC. The investor thus has a positive balance of 10,000, the current price. The collateralization ratio is $10,000/9,000 = 111\%$, which is a margin percentage of 11% and is nearly the maximum amount of allowed leverage (10% margin).

This intuition works similarly for a short position. The investor has committed to sell at 10,000, which is a positive balance and is supplemented by the margin deposit of 1,000 (so total of 11,000). The investor's negative balance is the obligation to buy 1 BTC, currently worth 10,000. The collateralization ratio is $11,000/10,000$, which corresponds to a margin of 10%.

Let's now follow the mechanics of a short position when the underlying asset (BTC) increases in value by 5%. If the price of BTC increases to 10,500 (a 5% increase), the margin percentage becomes $(11,000/10,500) - 1 = 4.76\%$ and the short position becomes subject to liquidation. The paper net balance of the position is \$500, the incentive for the liquidator to close the position collect the balance. Exhibit J reviews the mechanics of a long position.

EXHIBIT J

1 BTC = 10,000 USDC
 initial margin = **10%**
 maintenance margin = **5%**



Open long position of
1 BTC at 100,000 USDC
 Offer 1,000 USDC as margin

Long Position

Long Balance (what you will get)	Short Balance (what you owe)	Margin
10,000 1 BTC	10,000 - 1,000 = 9,000 USDC	$\frac{10,000}{9,000} - 1 = 11\%$

Scenario A

BTC ↑ by 10% to 11,000

Long Balance	Short Balance
10,000 1 BTC	9,000

Margin $\frac{11,000}{9,000} - 1 = 22.2\%$



- Trader can withdraw USDC to bring margin towards 10%
- Trader can close position with **1000 USDC** profit, which is a ROI of 100%

Scenario B

BTC ↓ by -7.5% to 9,250

Long Balance	Short Balance
9,250 1 BTC	9,000

Margin $\frac{9,250}{9,000} - 1 = 2.8\%$



- Position is below 5% maintenance margin requirement
- Keeper liquidates position by selling 1 BTC and paying back 9,000
- Keeper keeps \$250 USDC as reward

The dYdX BTC perpetual futures contract allows investors to access BTC returns natively on the Ethereum blockchain, while being able to supply any ERC-20 asset as collateral. Perpetual futures are rising in popularity, and this functionality may continue to attract liquidity over time.

Traditional Finance Problem	dYdX Solution
<i>Centralized Control:</i> Borrowing and lending rates controlled by institutions.	dYdX rates are determined algorithmically based on clearly outlined, transparent formulas (often asset pool utilization rates).
<i>Limited Access:</i> Difficulty in accessing high yield USD investment opportunities or competitive borrowing as well as futures and derivative products. Access to capital for immediately profitable enterprises is limited.	Open ability to borrow or lend any supported assets at competitive algorithmically determined rates. Includes a perpetual futures contract that could synthetically support any asset. Free flash loans give developers access to large amounts of capital to capitalize on arbitrage or other profitable opportunities.
<i>Inefficiency:</i> Suboptimal rates for borrowing and lending due to inflated costs.	Algorithmically pooled and optimized interest rates. Free flash loans offered for immediate use cases.
<i>Lack of Interoperability:</i> Difficult to	Flash loans can immediately utilize the

repurpose funds within a financial instrument.	entirety of the AUM for outside opportunities without risk or loss to investors.
<i>Opacity:</i> Unclear collateralization of lending institutions.	Transparent collateralization ratios of borrowers are visible to the entire ecosystem.

6.3.3 Synthetix

Many traditional derivative products have a decentralized counterpart. DeFi, however, allows new types of derivatives because of smart contracts. [Synthetix](#) is developing such a new type of derivative.

Imagine creating a derivative cryptoasset, whose value is based on an underlying asset that is neither owned nor escrowed. Synthetix is one group whose primary focus is creating a wide variety of liquid synthetic derivatives. Its model is, at a high level, straightforward and novel. The company issues *Synths*, tokens whose prices are pegged to an underlying price feed and are backed by collateral. MakerDAO's DAI is also a synthetic asset. The price feeds come from the [Chainlink](#)'s decentralized oracles.²⁹ Synths can theoretically track any asset, long or short, and even levered positions. In practice, there is no leverage, and the main tracked assets are cryptocurrencies, fiat currencies, and gold.

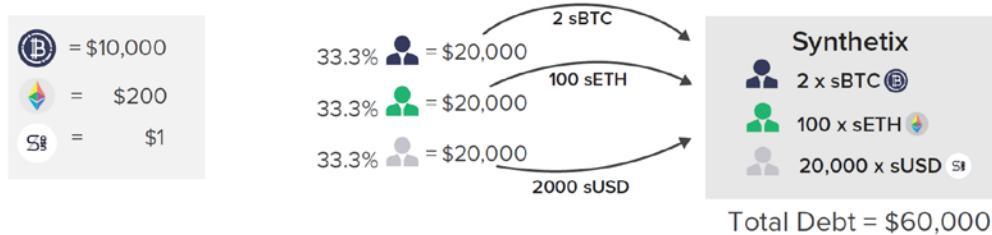
A long Synth is called an *sToken*, for example, a sUSD or a sBTC. The sUSD is a synthetic because its value is based on a price feed. A short Synth is called an *iToken*, for example, an iETH or an iMKR. Synthetix also has a platform token called SNX. SNX is not a governance token like MKR and COMP, but is a *utility token* or a *network token*, which means it enables the use of Synthetix functionality as its only feature. SNX serves as the unique collateral asset for the entire system. When users mint Synths against their SNX, they incur a debt proportioned to the total outstanding debt denominated in USD. They become *responsible* for this percentage of the debt in the sense that to unlock their SNX collateral they need to return the total USD value of their debt. The global debt of all Synths is thus shared collectively by the Synth holders based on the USD-denominated percentage of the debt they owned when they opened their positions. The total outstanding USD-denominated debt changes when any Synth's price fluctuates, and each holder remains responsible for the same percentage they were responsible for when they minted their Synths. Therefore, when a SNX holder's Synths outperform the collective pool, the holder effectively profits, and vice versa, because their asset value (their Synth position) outpaced the growth of the debt (sum of all sUSD debt).

As an example, three traders each have \$20,000 for a total debt of \$60,000: one holds 2 sBTC priced at \$10,000 each, one holds 100 sETH priced at \$200 each, and one holds 20,000 sUSD priced at \$1 each. Each has a debt proportion of 33.3%. If the price of BTC doubles to \$20,000 and the price of ETH spikes to \$1,000, the total debt becomes \$160,000 = \$40,000 (sBTC) +

²⁹ See <https://blog.synthetix.io/all-synths-are-now-powered-by-chainlink-decentralised-oracles/>

\$100,000 (sETH) + \$20,000 (sUSD).³⁰ Because each trader is responsible for 33.3%, about \$53,300, only the sETH holder is profitable even though the price of BTC doubled. If the price of BTC falls to \$5,000 and ETH to \$100, then the total debt falls to \$40,000 and the sUSD holder becomes the only profiting trader. Exhibit K details this example.

EXHIBIT K



Scenario 1

$\text{BTC} = \$20,000 (+100\%)$	$33.3\% \text{ (blue person)} \$40,000 - \$53,300 = -\$13,300$
$\text{ETH} = \$1,000 (+500\%)$	$33.3\% \text{ (green person)} \$100,000 - \$53,300 = \$46,700$
$\text{sUSD} = \$ (\text{no change})$	$33.3\% \text{ (grey person)} \$20,000 - \$53,300 = -\$33,300$
$\text{Debt} = \$160,000 / 3 = \$53,300$	

Synthetix
$2 \times \text{sBTC (BTC)} = \$ 40,000$
$100 \times \text{sETH (ETH)} = \$ 100,000$
$20,000 \times \text{sUSD (sUSD)} = \$ 20,000$
Total Debt = \$160,000

Scenario 2

$\text{BTC} = \$5,000 (-50\%)$	$33.3\% \text{ (blue person)} \$10,000 - \$13,300 = -\$3,300$
$\text{ETH} = \$100 (-50\%)$	$33.3\% \text{ (green person)} \$10,000 - \$13,300 = -\$3,300$
$\text{sUSD} = \$1 (\text{no change})$	$33.3\% \text{ (grey person)} \$20,000 - \$13,300 = +\$6,600$
$\text{Debt} = \$40,000 / 3 = \$13,300$	

Synthetix
$2 \times \text{sBTC (BTC)} = \$ 10,000$
$100 \times \text{sETH (ETH)} = \$ 10,000$
$20,000 \times \text{sUSD (sUSD)} = \$ 20,000$
Total Debt = \$40,000

The platform has a DEX native that will exchange any two Synths at the rate quoted by the oracle. SNX holders pay the exchange fees to a fee pool redeemable by SNX holders in proportion to their percentage of the debt. The contracts enforce that users can only redeem their fees if they maintain a sufficient collateralization ratio relative to their portion of the debt. The required collateralization ratio to mint Synths and participate in staking rewards is high, currently 800%. The Synthetix protocol also mints new SNX tokens via inflation to reward various stakeholders in the ecosystem for contributing value. The protocol distributes the rewards as a bonus incentive for maintaining a high collateralization ratio or increasing the liquidity of SNX.

³⁰ In any Synthetix position, the trader is effectively “long” his personal portfolio against the entire pool’s portfolio. In other words, the trader is betting his returns will exceed the pool’s returns. For example, by holding sUSD only, the trader is effectively shorting the entire composition of all other traders’ Synthetix portfolios and betting that USD will outperform all other assets held. The trader’s goal is to own Synths that he thinks will outperform the rest of the market, because it is the only way to profit.

Synthetix has branched into products that track real-world equities with the release of sNIKKEI and sFTSE. The protocol is also beginning to offer a binary options trading interface, further expanding its capabilities. The platform could easily gain popularity because there is no slippage against the price feed, however, the pooled liquidity and shared debt models offer interesting challenges.

Traditional Finance Problem	Synthetix Solution
<i>Centralized Control:</i> Assets can generally only be bought and sold on registered exchanges.	Offer synthetic assets in one place that can track any real world asset.
<i>Limited Access:</i> Access to certain assets is geographically limited.	Anyone can access Synthetix to buy and sell Synths. Some restrictions may eventually apply to those Synths that are securities.
<i>Inefficiency:</i> Large asset purchases suffer from slippage as traders eat into the liquidity pool.	Synths exchange rates are backed by a price feed, which eliminates slippage.
<i>Lack of Interoperability:</i> Real-world assets such as stocks can't be easily represented directly on a blockchain	Synth representations of real assets are totally compatible with Ethereum and other DeFi protocols.
<i>Opacity:</i> Lack of transparency in traditional derivative markets.	All protocol-based projects and features are transparently funded and voted upon by a DAO

6.4 Tokenization

Tokenization refers to the process of taking some asset or bundle of assets, either on or off chain, and

1. representing that asset on chain with possible fractional ownership, or
2. creating a composite token that holds some number of underlying tokens.

A token can conform to different specifications based on the type of properties a user wants the token to have. As mentioned earlier, the most popular token standard is ERC-20, the fungible token standard. This interface defines abstractly how a token, which has units that are non-unique and interchangeable (such as USD), should behave. An alternative is the ERC-721 standard, which defines nonfungible tokens (NFTs). These tokens are unique, such as a token representing ownership of a piece of fine art or a specific digital asset from a game. DeFi applications can take advantage of these and other standards to support any token using the standard simply by coding for the single standard.

6.4.1 Set Protocol

Set Protocol offers the “composite token” approach to tokenization. Instead of tokenizing assets non-native to Ethereum, Set Protocol combines Ethereum tokens into composite tokens that function more like traditional exchange traded funds (ETFs). Set Protocol combines cryptoassets into *Sets*, which are themselves ERC-20 tokens and fully collateralized by the components escrowed in a smart contract. A Set token is always redeemable for its components. Sets can be static or dynamic, based on a trading strategy. Static Sets are straightforward to understand and are simply bundled tokens the investor cares about; the resulting Set can be transferred as a single unit.

Dynamic Sets define a trading strategy that determines when reallocations can be made and at what times. Some examples include the “Moving Average” Sets that shift between 100% ETH and 100% USDC whenever ETH crosses its X-day simple or exponentially weighted moving average. Similar to normal ETFs, these Set tokens have fees and sometimes performance-related incentives. At the Set’s creation, the manager pre-programs the fees, which are paid directly to the manager for that particular Set. The available fee options are a buy fee (front-end load fee), a streaming fee (management fee), and a performance fee (percentage of profits over a high-water mark). The Set Protocol currently takes no fee for itself, although it may add a fee in the future. The prices and returns for Set Protocol are calculated via MakerDAOs’ publicly available oracle price feeds, which are also used by Synthetix. The main value-add of dynamic Sets is that the trading strategies are publicly encoded in a smart contract so users know exactly how their funds are being allocated and can easily redeem at any time.

Set Protocol also has a *Social Trading* feature in which a user can purchase a Set whose portfolio is restricted to certain assets with reallocations controlled by a single trader. Because these portfolios are actively managed, they function much more like mutual funds. The benefits are similar in that the portfolio manager has a predefined set of assets to choose from, and the users benefit from this contract-enforced transparency.

For example, a portfolio manager for a Set has a goal to “buy low and sell high” on ETH. The only assets she can use are ETH and USDC, and the only allocations she is allowed are 100% ETH and 100% USDC. At her sole discretion, she can trigger a contract function to rebalance the portfolio entirely into one asset or the other; this is the only allocation decision she can make. Assume she starts with 1,000 USDC. The price of ETH dips to 100 USDC/ETH and she decides to buy. She can trigger a rebalance to have 10 ETH in the Set. If the price of ETH doubles to \$200, the entire Set is now worth \$2,000. A shareholder who owns 10% of the Set can redeem her shares for 1 ETH.

Sets could democratize wealth management in the future by being more peer to peer, allowing fund managers to gain investment exposures through nontraditional channels and giving all investors access to the best managers. A further enhancement many Sets take advantage of is that their components use cTokens, the Compound-invested version of tokens. Between rebalances, tokens earn interest through the Compound protocol. This is one example of DeFi platforms being composed to create new products and value for investors.

Traditional Finance Problem	Set Protocol Solution
<i>Centralized Control:</i> Fund managers can control their funds against the will of investors.	Enforces sovereignty of the investor over their funds at the smart contract level.
<i>Limited Access:</i> Talented fund managers often are unable to gain exposures and capital to run a successful fund.	Allows anyone to become a fund manager and display their skills using social trading features.
<i>Inefficiency:</i> Many arising from antiquated practices.	Trading strategies encoded in smart contracts lead to optimal execution.
<i>Lack of Interoperability:</i> Difficult to combine assets into new packages and incorporate the combined assets into new financial products.	Set tokens are ERC-20 compliant tokens that can be used on their own in other DeFi protocols. For example, Aave allows Set token borrowing and lending for some popular Sets.
<i>Opacity:</i> Difficult to know the breakdown of assets in an ETF or mutual fund at any given time.	Total transparency into strategies and allocations of Set tokens.

6.4.2 wBTC

The [wBTC](#) application takes the *representing off-chain assets on chain* approach to tokenization, specifically for BTC. Abstractly, wBTC allows BTC to be included as collateral or liquidity on all of the Ethereum-native DeFi platforms. Given that BTC has comparatively low volatility and is the most well-adopted cryptocurrency by market-cap, this characteristic unlocks a large potential capital pool for DeFi dApps.

The wBTC ecosystem contains three key stakeholders: users, merchants, and custodians. Users are simply the traders and DeFi participants who generate demand for the value proposition associated with wBTC, namely, Ethereum-tokenized BTC. Users can purchase wBTC from merchants by transferring BTC and performing the requisite KYC/AML, thus making the entry and exit points of wBTC centralized and reliant on off-chain trust and infrastructure. Merchants are responsible for transferring wBTC to the custodians. At the point of transfer, the merchant signals to an on-chain Ethereum smart contract that the custodian has taken custody of the BTC and is approved to mint wBTC. Custodians use industry-standard security mechanisms to custody the BTC until it is withdrawn from the wBTC ecosystem. Once the custodians have confirmed receipt, they can trigger the minting of wBTC that releases wBTC to the merchant. Finally, closing the loop, the merchant transfers the wBTC to the user.

No single participant can control the minting and burning of wBTC, and all BTC entering the system is audited via transaction receipts that verify custody of on-chain funds. These safeguards increase the system's transparency and reduce the risk to users that is inherent in the system.

Because the network consists of merchants and custodians, any fraud is quickly expungeable from the network at only a small overall cost versus the cost that would be incurred in a single centralized entity. The mechanism by which merchants and custodians enter and leave the network is a multi-signature wallet controlled by the wBTC DAO. In this case, the DAO does not have a governance token; instead, a set of owners who can add and remove owners controls the DAO. The contract currently allows a maximum of 50 owners, with a minimum threshold of 11 to invoke a change. The numbers 50 and 11 can be changed, if the number of conditions are met. This system is more centralized than other governance mechanisms we have discussed, but is still more decentralized than allowing a single custodian to control all of the wBTC.

7. Risks

As we have emphasized in previous sections, DeFi allows developers to create new types of financial products and services, expanding the possibilities of financial technology. While DeFi can eliminate counterparty risk, cutting out middlemen and allowing financial assets to be exchanged in a trustless way, as with any innovative technology, the innovations introduce a new set of risks. In order to provide users and institutions with a robust and fault-tolerant system capable of handling new financial applications at scale, we must confront these risks. Without proper risk mitigation, DeFi will remain an exploratory technology, restricting its use, adoption, and appeal.

The principal risks DeFi faces today are smart contract, governance, oracle, scaling, exchange, custodial and regulatory risks.

7.1 Smart-Contract Risk

Over the past decade, crypto-focused products, primarily exchanges, have repeatedly been hacked. Whereas many of these hacks happened because of poor security practices, they demonstrate an important point: software is uniquely vulnerable to hacks and developer malpractice. Blockchains can remove traditional financial risks, such as counterparty risk, with their unique properties, but DeFi is built on code. This software foundation gives attackers a larger attack surface than the threat vectors of traditional financial institutions. As we discussed previously, public blockchains are open systems. Anyone can view and interact with code on a blockchain after the code is deployed. Given that this code is often responsible for storing and transferring blockchain native financial assets, it introduces a new, unique risk. This new attack vector is termed *smart contract risk*.

So what does smart contract risk mean?

DeFi's foundation is public computer code known as a smart contract. While the concept of a smart contract was first introduced by Nick Szabo in [his 1997 paper](#), the implementation is new to mainstream engineering practice. Therefore, formal engineering practices that will help reduce the risk of smart contract bugs and programming errors are still under development. The recent

hacks of [DForce](#) and [bZx](#)³¹ demonstrate the fragility of smart contract programming, and auditing firms, such as [Quantstamp](#), [Trail of Bits](#), and [Peckshield](#), are emerging to fill this gap in best practices and smart contract expertise.

Smart Contract risk can take the form of a logic error in the code or an economic exploit in which an attacker can withdraw funds from the platform beyond the intended functionality. The former can take the form of any typical software bug in the code. For example, let's say we have a smart contract which is intended to be able to escrow deposits from a particular ERC-20 from any user and transfer the entire balance to the winner of a lottery. The contract keeps track of how many tokens it has internally, and uses that internal number as the amount when performing the transfer. The bug will belong here in our hypothetical contract. The internal number will, due to a rounding error, be slightly higher than the actual balance of tokens the contract holds. When it tries to transfer, it will transfer "too much" and the execution will fail. If there was no failsafe put into place, the tokens are functionally locked within the protocol. Informally these are known as "bricked" funds and cannot be recovered.

An economic exploit would be more subtle. There would be no explicit failure in the logic of the code, but rather an opportunity for an economically equipped adversary to influence market conditions in such a way as to profit inappropriately at the contract's expense. For example, let's assume a contract takes the role of an exchange between two tokens. It determines the price by looking at the exchange rate of another similar contract elsewhere on chain and offering that rate with a minor adjustment. We note here that the other exchange is playing the role of a price oracle for this particular contract. The possibility for an economic exploit arises when the oracle exchange has significantly lower liquidity when compared to the primary exchange in our example. A financially equipped adversary can purchase heavily on the oracle exchange to manipulate the price, then proceed to purchase far more on the primary exchange in the opposite direction to capitalize on the price movement. The net effect is that the attacker was able to manufacture a discounted price on a high liquidity exchange by manipulating a low liquidity oracle.

Economic exploits become even trickier when considering that flash loans essentially allow any Ethereum user to become financially equipped for a single transaction. Special care must be used when designing protocols such that they cannot be manipulated by massive market volatility within a single transaction. An economic exploit which utilizes a flash loan can be referred to as a *flash attack*. A series of high profile flash attacks were executed in Feb 2020 on bZx Fulcrum, a lending market similar to Compound.³² The attacker utilized a flash loan and diverted some of the funds to purchase a levered short position, with the remainder used to manipulate the price of the oracle exchange which the short position was based on. The attacker then closed the short at a profit, unwound the market trade and paid back the flash loan. The net profit was almost \$300,000 worth of funds previously held by bZx, for near zero upfront cost.

³¹ See <https://cointelegraph.com/news/dforce-hacker-returns-stolen-money-as-criticism-of-the-project-continues> and <https://cointelegraph.com/news/decentralized-lending-protocol-bzx-hacked-twice-in-a-matter-of-days>

³² <https://bzx.network/blog/postmortem-ethdenver>

The most famous smart contract attack occurred in 2016. A smart contract was designed by Slock.it to act as the first decentralized venture capital fund for blockchain ventures. It was launched in April 2016³³ and attracted about 14% of all the ether available at the time. The DAO tokens began trading in May. However, there was a crucial part of the code with two lines in the wrong order allowing the withdrawal of ether repeatedly - before checking to see if the hacker was entitled to withdraw. This flaw is known as the reentrancy bug. On June 17, a hacker drained 30% of the value of the contract before another group, the Robin Hood Group, drained the other 70%. The Robin Hood Group promised to return all the ether to the original owners.

The original contract had a 28-day hold period before the funds could be withdrawn. The Ethereum community debated whether they should rewrite history by creating a hard fork (which would eliminate the hack). In the end, they decided to do the hard fork and returned the ether to the original investors. The old protocol became Ethereum Classic (ETC) which preserved the immutable record. The initiative halted in July when the SEC declared that DAO tokens were securities.

There have been many exploits like this. In April 2020, hackers exploited \$25m from dForce's Lendf.Me lending protocol. Interestingly, the Lendf.Me code was largely copied from Compound. Indeed, the word "Compound" appears four times in dForce's contract. The CEO of Compound remarked: "If a project doesn't have the expertise to develop its own smart contracts, ... it's a sign that they don't have the capacity or intention to consider security."³⁴

A smaller but fascinating attack occurred in February 2021 and the target was Yearn.finance.³⁵ Yearn is a yield aggregator. Users deposit funds into pools and these funds are allocated to other DeFi protocols to maximize the yield for the original investors. The transaction included 161 token transfers using Compound, dYdX, Aave, Uniswap and cost over \$5,000 in gas fees.³⁶ It involved flash loans of over \$200m.

Smart-contract programming still has a long way to go before best practices are developed and complex smart-contracts have the resilience necessary to handle high-value transactions. As long as smart-contract risk threatens the DeFi landscape, application adoption and trust will suffer as users hesitate to trust the contracts they interact with and that custody their funds.

7.2 Governance Risk

Programming risks are nothing new. In fact, they have been around since the dawn of modern computing more than half a century ago. For some protocols, such as Uniswap, programming risk is the sole threat to the protocol because the application is autonomous and controlled by smart contracts. Other DeFi applications rely on more than just autonomous computer code. For

³³ Ethereum block 1428757.

³⁴ <https://decrypt.co/26033/dforce-lendfme-defi-hack-25m>

³⁵ <https://www.theblockcrypto.com/linked/93818/yearn-finance-dai-pool-defi-exploit-attack>

³⁶ <https://etherscan.io/tx/0x6dc268706818d1e6503739950abc5ba221fc6b451e54244da7b1e226b12e027>

example, MakerDAO, the decentralized credit facility described earlier, is reliant on a human-controlled governance process that actively adjusts protocol parameters to keep the system solvent. Many other DeFi protocols use similar systems and rely on humans to actively manage protocol risk. This introduces a new risk, *governance risk*, which is unique to the DeFi landscape.

Protocol governance refers to the representative or liquid democratic mechanisms that enable changes in the protocol.³⁷ To participate in the governance process, users and investors must acquire a token that has been explicitly assigned protocol governance rights on a liquid marketplace. Once acquired, holders use these tokens to vote on protocol changes and guide future direction. Governance tokens usually have a fixed supply that assists in resisting attempts by anyone to acquire a majority (51%), nevertheless they expose the protocol to the risk of control by a malicious actor. While we have yet to see a true governance attack in practice, new projects like Automata³⁸ allow users to buy governance votes directly, and will likely accelerate the threat of malicious/hostile governance.

The founders often control traditional fintech companies, which reduces the risk of an external party influencing or changing the company's direction or product. DeFi protocols, however, are vulnerable to attack as soon as the governance system launches. Any financially equipped adversary can simply acquire a majority of liquid governance tokens to gain control of the protocol and steal funds.

On March 13, 2021 there was a governance attack on True Seigniorage Dollar. In this particular situation, the developers controlled only 9% of the DAO. The attacker gradually bought \$TSD until he had 33% of the DAO. The hacker then proposed an implementation and voted for it. The attacker added code to mint himself 11.5 quintillion \$TSD and then sold 11.8b \$TSD tokens on Pancakeswap.³⁹

We have not yet experienced a successful governance attack on any Ethereum-based DeFi project, but little doubt exists that a financially equipped adversary will eventually attack a protocol if the potential profit exceeds the cost of attack.

7.3 Oracle Risk

Oracles are one of the last unsolved problems in DeFi and are required by most DeFi protocols in order to function correctly. Fundamentally, oracles aim to answer the simple question: How can off-chain data be securely reported on chain? Without oracles, blockchains are completely self-encapsulated and have no knowledge of the outside world other than the transactions added to the native blockchain. Many DeFi protocols require access to secure, tamper-resistant asset prices to ensure that routine actions, such as liquidations and prediction market resolutions, function correctly. Protocol reliance on these data feeds introduces *oracle risk*.

³⁷ <https://medium.com/dragonfly-research/decentralized-governance-innovation-or-imitation-ad872f37b1ea>

³⁸ <https://automata.fi/>

³⁹ <https://twitter.com/trueseigniorage/status/1370956726489415683?lang=en>

Oracles represent significant risks to the systems they help support. If an oracle's *Cost of Corruption* is ever less than an attacker's potential *Profit from Corruption*, the oracle is extremely vulnerable to attack.

To date, three types of oracle solutions have been introduced, developed, and used. The first is a *Schelling-point oracle*. This oracle relies on the owners of a fixed-supply token to vote on the outcome of an event or report the price of an asset. Examples of this type of oracle include [Augur](#) and [UMA](#). While Schelling-point oracles preserve the decentralization components of protocols that rely on them, they suffer from slow times to resolution.

The second type of oracle solution an *API oracle*. These oracles are centralized entities that respond asynchronously to requests for data or prices. Examples include [Provable](#), [Oraclize](#), and [Chainlink](#). All systems relying on API-based oracles, must trust the data provider to respond accurately to all queries.

The third type of oracle is a custom, application-specific oracle service. This type of oracle is used by Maker and Compound. Its design differs based on the requirements of the protocol it was developed for. For example, Compound relies on a single data provider that the Compound team controls to provide all on-chain price data to the Compound oracle.

Oracles, as they exist today, represent the highest risk to DeFi protocols that rely on them. All on-chain oracles are vulnerable to [front-running](#), and [millions of dollars](#) have been lost due to arbitrageurs. Additionally, oracle services, including [Chainlink](#) and Maker, have suffered [crippling outages](#) with catastrophic downstream effects.

Until oracles are blockchain native, hardened, and proven resilient, they represent the largest systemic threat to DeFi today.

7.4 Scaling Risk

As we have discussed, Ethereum and other "Proof of Work" (the consensus mechanism) blockchains have a fixed block size. For a block to become part of the chain, every Ethereum miner must execute all of the included transactions on their machine. To expect each miner to process all of the financial transactions for a global financial market is unrealistic. Ethereum is currently limited to a maximum of 15 TPS. Yet, almost all of DeFi today resides on this blockchain. Compared to Visa, which can handle upward of 65,000 transactions per second, Ethereum is capable of handling less than 0.1% of the throughput. Ethereum's lack of scalability places DeFi at risk of being unable to meet requisite demand. Much effort is focused on increasing Ethereum's scalability or replacing Ethereum with an alternative blockchain that can more readily handle higher transaction volumes. To date, all efforts have proven unsuccessful for Ethereum. However, some new platforms such as [Polkadot](#), [Zilliqa](#) and [Algorand](#) offer some solutions for this scaling risk.

One actively pursued solution to the problem is a new consensus algorithm, *Proof of Stake*. Proof of Stake simply replaces mining of blocks (which requires a probabilistic wait time), with staking an asset on the next block, with majority rules similar to PoW.

Staking, an important concept in cryptocurrencies and DeFi, means a user escrows funds in a smart contract and is subject to a penalty (*slashed funds*) if they deviate from expected behavior.

An example of malicious behavior in Proof of Stake includes voting for multiple candidate blocks. This action shows a lack of discernment and skews voting numbers, and thus is penalized. The security in Proof of Stake is based on the concept that a malicious actor would have to amass more of the staked asset (ether in the case of Ethereum) than the entire rest of the stakers on that chain. This goal is infeasible and hence results in strong security properties similar to PoW.

Vertical and horizontal scaling are two additional general approaches to increasing blockchain throughput. Vertical scaling centralizes all transaction processing to a single large machine. This centralization reduces the communication overhead (transaction/block latency) associated with a PoW blockchain such as Ethereum, but results in a centralized architecture in which one machine is responsible for a majority of the system's processing. Some blockchains, such as [Solana](#), follow this approach and can achieve upward of 50,000 TPS.

Horizontal scaling, however, divides the work of the system into multiple pieces, retaining decentralization but increasing the throughput of the system through parallelization. *Ethereum 2.0* takes this approach (called *sharding*) in combination with a Proof of Stake consensus algorithm.

Ethereum 2.0's technical architecture⁴⁰ differs drastically from vertically scaled blockchains such as Solana, but the improvements are the same. Ethereum 2.0 uses horizontal scaling with multiple blockchains and can achieve upward of 50,000 transactions per second.

The development of Ethereum 2.0 has been delayed for several years, but its mainnet, which will contain a basic blockchain without any smart contract support, may go live in 2021. Ethereum 2.0 has not yet finalized a functional specification for sending transactions between its horizontally scaled blockchains.

Another competitor with the potential to reduce scaling risk is the Ethereum layer-2 landscape. *Layer 2* refers to a solution built on top of a blockchain that relies on cryptography and economic guarantees to maintain desired levels of security. Transactions can be signed and aggregated in a form resistant to malicious actors, but are not directly posted to the blockchain unless there is a discrepancy of some kind. This removes the constraints of a fixed block size and block rate, allowing for much higher throughput. Some layer-2 solutions are live today.

As Ethereum's transaction fees have risen to record levels, layer-2 usage has remained stagnant. The space has been developing slowly and many live solutions lack support for smart contracts or decentralized exchanges. One solution in development is an *Optimistic Rollup*. An optimistic

⁴⁰ See <https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/>

rollup is a process in which transactions are aggregated off-chain into a single digest that is periodically submitted to the chain over a certain interval. Only an aggregator who has a bond (stake) can combine and submit these summaries. Importantly, the state is assumed to be valid unless someone challenges it. If a challenge occurs, cryptography can prove if the aggregator posted a faulty state. The prover is then rewarded with a portion of the malicious aggregator's bond as an incentive (similar to a Keeper mechanism). Optimistic rollups have yet to deliver functional mainnets and require expensive fraud proofs as well as frequent rollup transaction posting, limiting their throughput and increasing their average transaction costs.

Many approaches aim to decrease the scalability risks facing DeFi today, but the field lacks a clear winner. As long as DeFi's growth is limited by blockchain scaling, applications will be limited in their potential impact.

7.5 DEX Risk

The most popular DeFi products today mirror those we observe in traditional finance. The main uses for DeFi are gaining leverage, trading, and acquiring exposure to synthetic assets. Trading, as might be expected, accounts for the highest on-chain activity, while the introduction of new assets (ERC-20 tokens, Synthetics, and so forth) has led to a Cambrian explosion in DEXs. These decentralized exchanges vary considerably in design and architecture, but all are attempts to solve the same problem—how to create the best decentralized venue to exchange assets?

The DEX landscape on Ethereum consists of two dominant types, Automated Market Makers (AMMs) and order-book exchanges. Both types of DEXs vary in architecture and have differing risk profiles. AMMs, however, are the most popular DEX to date, because they allow users to trustlessly and securely exchange assets, while removing traditional counterparty risk. By storing exchange liquidity in a trustless smart contract, AMMs give users instant access to quotes on an exchange pair. Uniswap is perhaps the best-known example of an AMM, also known as a Constant-Function Market Maker (CFMM). Uniswap relies on the product of two assets to determine an exchange price (see section 7.3). The amount of liquidity in the pool determines the slippage when assets are exchanged during a transaction.

CFMMs such as Uniswap optimize for user experience and convenience, but sacrifice absolute returns. CFMM liquidity providers (LPs) earn yield by depositing assets into a pool, because the pool takes a fee for every trade (LPs benefit from high trading volume). This allows the pool to attract liquidity, but exposes LPs to smart contract risk and impermanent loss. Impermanent loss occurs when two assets in a pool have uncorrelated returns and high volatilities.⁴¹ These properties allow arbitrageurs to profit from the asset volatilities and price differences, reducing the temporary returns for LPs and exposing them to risk if an asset moves sharply in price. Some AMMs, such as [Cap](#), are able to reduce impermanent loss by using an oracle to determine exchange prices and dynamically adjusting a pricing curve to prevent arbitrageurs from exploiting LPs, but impermanent loss remains a large problem with most AMMs used today.

⁴¹ For more on this topic, see Qureshi (2020).

On-chain order-book DEXs have a different but prevalent set of risks. These exchanges suffer from the scalability issues inherited from the underlying blockchain they run atop of, and are often vulnerable to front running by sophisticated arbitrage bots. Order-book DEXs also often have large spreads due to the presence of low-sophistication market makers. Whereas traditional finance is able to rely on sophisticated market makers including [Jump](#), [Virtu](#), [DRW](#), [Jane Street](#) and more, order-book DEXs are often forced to rely on a single market maker for each asset pair. This reliance is due to the nascentcy of the DeFi market as well as the complex compute infrastructure required to provide on-chain liquidity to order-book DEXs. As the market evolves, we expect these barriers to break down and more traditional market makers to enter the ecosystem; for now, however, these obstacles create a significant barrier to entry. Regardless, both AMM and order-book DEXs are able to eliminate counterparty risk while offering traders a noncustodial and trustless exchange platform.

Several decentralized exchanges use an entirely off-chain order book, retaining the benefits of a noncustodial DEX, while circumventing the market making and scaling problems posed by on-chain order-book DEXs. These exchanges function by settling all position entries and exits on chain, while maintaining a limit-order book entirely off chain. This allows the DEX to avoid the scaling and UX issues faced by on-chain order-book DEXs, but also presents a separate set of problems around regulatory compliance.

Although risks abound in the DEX landscape today, they should shrink over time as the technology advances and market players increase in sophistication.

7.6 Custodial Risk

There are three types of custody: self, partial, and third-party custody. With self custody, a user develops their own solution which might be a flash drive not connected to the internet, a hard copy, or a vaulting device. With partial custody, there is a combination of self custody and external solution (e.g., Bitgo). Here, a hack on the external provider provides insufficient information to recreate the private key. However, if the user loses their private key, the user combined with the external solution, can recreate the key. The final option is third-party custody. There are many companies that have traditionally focused on custody in centralized finance that are now offering solutions in decentralized finance (e.g., Fidelity Digital Assets).

Retail investors generally face two options. The first is self custody where users have full control over their keys. This includes a hardware wallet, web wallet (e.g., MetaMask where keys are stored in a browser), desktop wallet, or even a paper wallet. The second is a custodial wallet. Here a third party holds your private keys. Examples are Coinbase and Binance.

The most obvious risk for self custody is that the private keys are lost or locked. In January 2021, The New York Times ran a story about a programmer who used a hardware wallet but forgot the

password.⁴² The wallet contains \$220m of bitcoin. The hardware wallet allows 10 password attempts before all data are destroyed. The programmer has only two tries to go.

Delegated custody also involves risks. For example, if an exchange holds your private keys, the exchange could be hacked and your keys lost. Most exchanges keep the bulk of private keys in “cold storage” (on a drive not connected to the internet). Nevertheless, there is a long history of exchange attacks including: Mt Gox (2011-2014) 850,000 bitcoin, Bitfloor (2012) 24,000 bitcoin, Bitfinex (2016) 120,000 bitcoin, Coincheck (2018) 523m NEM worth \$500m at the time, and Binance (2019) 7,000 bitcoin.⁴³ The attacks have become less frequent. Some exchanges, such as Coinbase, even offer insurance. All of these attacks were on centralized exchanges. We have already reviewed some of the attacks on DEXs.

7.7 Regulatory Risk

As the DeFi market increases in size and influence, it will face greater regulatory scrutiny. Major centralized spot and derivatives exchanges, previously ignored by the CFTC, have recently been forced to comply with [KYC/AML compliance orders](#), and DEXs appear to be next. Already, several decentralized derivatives exchanges, such as dYdX, must geoblock US customers from accessing certain exchange functionalities. Whereas the noncustodial and decentralized nature of DEXs presents a legal grey area with an uncertain regulatory future, little doubt exists that regulation will arrive once the market expands.

A well known algorithmic stablecoin project known as [Basis](#) was forced to shut down in December of 2018 due to regulatory concerns. A harrowing message remains on their home page for future similar companies: “Unfortunately, having to apply US securities regulation to the system had a serious negative impact on our ability to launch Basis...As such, I am sad to share the news that we have decided to return capital to our investors. This also means, unfortunately, that the Basis project will be shutting down.” In response to regulatory pressure, DeFi has seen an increasing number of anonymous protocol founders. Earlier this year, an anonymous team launched a fork of the original Basis project ([Basis Cash](#)⁴⁴).

Governance tokens, released by many DeFi projects, are also facing increasing scrutiny as the SEC continues to evaluate if these new assets will be regulated as securities. For example, Compound, the decentralized money market on Ethereum, recently released a governance token with no intrinsic value or rights to future cash flows. Doing so allowed Compound to avoid the SEC’s securities regulation, freeing the company from security issuance responsibilities. We predict more projects will follow Compound’s example in the future, and we expect most to exercise caution before issuing new tokens; many projects learned from the harsh [penalties](#) the SEC issued following the ICO boom in 2017.

⁴² <https://www.nytimes.com/2021/01/12/technology/bitcoin-passwords-wallets-fortunes.html>

⁴³ <https://blog.idex.io/all-posts/a-complete-list-of-cryptocurrency-exchange-hacks-updated>

⁴⁴ <https://basis.cash/>

Many major market-cap cryptocurrencies have been ruled commodities by the CFTC, exempting them from money-transmitter laws. Individual states, such as [New York](#), however, have regulation that targets brokerages facilitating the transfer and exchange of cryptocurrencies. As DeFi continues to grow and the total number of issued assets continues to expand, we expect to see increasingly specific and nuanced regulation aimed at DeFi protocols and their users.

Cryptocurrency taxation has yet to be fully developed from a regulatory standpoint, and accounting software/on-chain monitoring is just starting to reach mainstream retail audiences. For example, as of December 31, 2020, the IRS draft proposal requires reporting on form 1040 of: any receipt of cryptocurrency (for free) including airdrop or hard fork; exchange of cryptocurrency for goods or services; purchase or sale of cryptocurrency; exchange of virtual currency for other property, including for another virtual currency; and acquisition or disposition of a financial interest in a cryptocurrency. Moving virtual currency from one wallet to another is not included. The regulations also make it clear that if you received any cryptocurrency for work, that must be reported on form W2.⁴⁵

While the DeFi regulatory landscape continues to be actively explored, with new regulatory decisions being made daily such as that allowing [banks to custody cryptocurrency](#), the market outlook is hazy with many existing problems yet to be navigated.

8. Conclusions: The Losers and the Winners

Decentralized finance provides compelling advantages over traditional finance along the verticals of decentralization, access, efficiency, interoperability, and transparency. Decentralization allows financial products to be owned collectively by the community without top-down control, which could be hazardous to the average user. Access to these new products for all individuals is of critical importance in preventing widening wealth gaps.

Traditional finance exhibits layers of fat and inefficiency that ultimately remove value from the average consumer. The contractual efficiency of DeFi brings all of this value back. As a result of its shared infrastructure and interfaces, DeFi allows for radical interoperability beyond what could ever be achieved in the traditional-finance world. Finally, the public nature of DeFi fosters trust and security where there may traditionally exist opacity.

DeFi can even directly distribute value to users to incentivize its growth, as demonstrated by Compound (via COMP) and Uniswap (via UNI). *Yield farming* is the practice of seeking rewards by depositing into platforms that incentivize liquidity provisioning. Token distributions and yield farming have attracted large amounts of capital to DeFi over incredibly short time windows.

⁴⁵ <https://www.irs.gov/pub/irs-dft/f1040gi--dft.pdf>

Platforms can engineer their token economics to both reward their innovation and foster a long-term sustainable protocol and community that continues to provide value.

Each DeFi use case embodies some of these benefits more than others and has notable drawbacks and risks. For example, a DeFi platform, which heavily relies on an oracle that is more centralized, can never be as decentralized as a platform that needs no external input to operate, such as Uniswap. Additionally, a platform such as dYdX with some off-chain infrastructure in its exchange cannot have the same levels of transparency and interoperability.

Certain risks plague all of DeFi and overcoming them is crucial to DeFi's achieving mainstream adoption. Two risks, in particular, are scaling risk and smart contract risk. The benefits of DeFi will be limited to only the wealthiest parties if the underlying technology cannot scale to serve the population at large. Inevitably, the solutions to the scaling problem will come at the price of some of the benefits of a "pure" DeFi approach, such as decreased interoperability on a "sharded" blockchain. Similar to the internet and other transformational technologies, the benefits and scale will improve over time. Smart contract risk will never be eliminated, but wisdom gained from experience will inform best practices and industry trends going forward.

As a caution to dApps that blindly integrate and stack on top of each other without proper due diligence, the weakest link in the chain will bring down the entire house. The severity of smart contract risk grows directly in proportion to the natural tendency to innovate and integrate with new technologies. For this reason, it is inevitable that high-profile vulnerabilities will continue to jeopardize user funds as they have in the past. If DeFi cannot surmount these risks, among others, its utility will remain a shadow of its potential.

The true potential of DeFi is transformational. Assuming DeFi realizes its potential, the firms that refuse to adapt will be lost and forgotten. All traditional finance firms can and should begin to integrate their services with crypto and DeFi as the regulatory environment gains clarity and the risks are better understood over time. This adoption can be viewed as a "DeFi front end" that abstracts away the details to provide more simplicity for the end user.

Startups, such as [Dharma](#), are leading this new wave of consumer access to DeFi. This approach will still suffer from added layers of inefficiency, but the firms that best adopt the technology and support local regulation will emerge as victors while the others fade away. The DeFi protocols that establish strong liquidity moats and offer the best utility will thrive as the key backend to mainstream adoption.

We see the scaffolding of a shiny new city. This is not a renovation of existing structures; it is a complete rebuild from the bottom up. Finance becomes accessible to all. Quality ideas are funded no matter who you are. A \$10 transaction is treated identically to a \$100m transaction. Savings rates increase and borrowing costs decrease as the wasteful middle layers are excised. Ultimately, we see DeFi as the greatest opportunity of the coming decade and look forward to the reinvention of finance as we know it.

9. References

- Chetty, Raj, Nathaniel Hendren, Patrick Kline, and Emmanuel Saez. 2014. "Where Is the Land of Opportunity? The Geography of Intergenerational Mobility in the United States." *Quarterly Journal of Economics*, vol. 129, no. 4 (November): 1553–1623.
- Corbae, Dean, and Pablo D'Erasmo. 2020. "Rising Bank Concentration," Staff Paper 594, Federal Reserve Bank of Minneapolis (March). Available at <https://doi.org/10.21034/sr.594>
- Ellis, Steve, Ari Juels, and Sergey Nazarov. 2017. "Chainlink: A Decentralized Oracle Network." Working paper (September 4). Available at <https://link.smartcontract.com/whitepaper>
- Euromoney*. 2001. "Forex Goes into Future Shock." (October). Available at <https://faculty.fuqua.duke.edu/~charvey/Media/2001/EuromoneyOct01.pdf>
- Haber, Stuart, and Scott Stornetta. 1991. "How to Time-Stamp a Digital Document." *Journal of Cryptology* (January). Available at <https://dl.acm.org/doi/10.1007/BF00196791>
- Nakamoto, Satoshi. 2008. "Bitcoin: A Peer-to-Peer Electronic Cash System." Bitcoin.org.
- Narayan, Amber, Roy Van der Weide, Alexandru Cojocaru, Christoph Lakner, Silvia Redaelli, Daniel Mahler, Rakesh Ramasubbaiah, and Stefan Thewissen. 2018. *Fair Progress? Economic Mobility across Generations around the World*, Equity and Development Series. Washington, DC: World Bank.
- Qureshi, Haseeb. 2020. "What Explains the Rise of AMMs?" Dragonfly Research (July 22).
- Ramachandran, Ashwin, and Haseeb Qureshi. 2020. "Decentralized Governance: Innovation or Imitation?" Dragonfly Research.com (August 5). Available at <https://medium.com/dragonfly-research/decentralized-governance-innovation-or-imitation-ad872f37b1ea>
- Robinson, Dan, and Allan Niemerg. 2020. "The Yield Protocol: On-Chain Lending with Interest Rate Discovery." White paper (April). Available at <https://research.paradigm.xyz/Yield.pdf>
- Shevchenko, Andrey. 2020. "Dforce Hacker Returns Stolen Money as Criticism of the Project Continues." Cointelegraph.com (April 22).
- Szabo, Nick. 1997. "Formalizing and Securing Relationships on Public Networks." Satoshi Nakamoto Institute. Available at <https://nakamotoinstitute.org/formalizing-securig-relationships/>
- Zmudzinski, Adrian. 2020. "Decentralized Lending Protocol bZx Hacked Twice in a Matter of Days." Cointelegraph.com (February 18).

10. Glossary

The italicized terms in the glossary definitions are themselves defined in the glossary.

Address. The address is the identifier where a transaction is sent. The address is derived from a user's public key. The public key is derived from the private key by *asymmetric key cryptography*. In Ethereum, the public key is 512 bits or 128 *hexadecimal* characters. The public key is hashed (i.e., uniquely represented) with a Keccak-256 algorithm, which transforms it into 256 bits or 64 hexadecimal characters. The last 40 hexadecimal characters are the public key. The public key usually carries the pre-fix "0x." Also known as public address.

Airdrop. Refers to a free distribution of tokens into wallets. For example, Uniswap governance airdropped 400 tokens into every Ethereum address that had used their platform.

AML (Anti-Money Laundering). A common compliance regulation designed to detect and report suspicious activity related to illegally concealing the origins of money.

AMM. See **Automated market maker**.

Asymmetric key cryptography. A means to secure communication. Cryptocurrencies have two keys: public (everyone can see) and private (secret and only for the owner). The two keys are connected mathematically in that the private key is used to derive the public key. With current technology, it is not feasible to derive the private key from the public key (hence, the description "asymmetric"). A user can receive a payment to their public address and spend it with their private key. Also, see *symmetric key cryptography*.

Atomic. A provision that causes contract terms to revert as if tokens never left the starting point, if any contract condition is not met. This provision is an important feature of a *smart contract*.

Automated market maker (AMM). A *smart contract* that holds assets on both sides of a trading pair and continuously quotes a price for buying and for selling. Based on executed purchases and sales, the contract updates the asset size behind both the bid and the ask and uses this ratio to define a pricing function.

Barter. A peer-to-peer exchange mechanism in which two parties are exactly matched. For example, A has two pigs and needs a cow. B has a cow and needs two pigs. There is some debate as to whether barter was the first method of exchange. For example, David Graeber argues that the earliest form of trade was in the form of debit/credit. People living in the same village gave each other "gifts" which by social consensus had to be returned in future by another gift that is usually a little more valuable (interest). People kept track of exchanges in their minds as it was only natural and convenient to do so since there is only a handful sharing the same village.

Coinage comes into play many, many years later with the rise of migration and war with war tax being one of the very first use cases.⁴⁶

Blockchain. A decentralized ledger invented in 1991 by Haber and Stornetta. Every *node* in the ledger has a copy. The ledger can be added to through *consensus protocol*, but the ledger's history is immutable. The ledger is also visible to anyone.

Bonding curve. A *smart contract* that allows users to buy or sell a token using a fixed mathematical model. For example, consider a simple linear function in which the token = supply. In this case, the first token would cost 1 ETH and the second token 2 ETH, thereby rewarding early participants. It is possible to have different bonding curves for buying and selling. A common functional form is a logistic curve.

Bricked funds. Funds trapped in a *smart contract* due to a bug in the contract.

Burn. The removal of a token from circulation, which thereby reduces the supply of the token. Burning is achieved by sending the token to an unowned *Ethereum* address or to a contract that is incapable of spending. Burning is an important part of many *smart contracts*. For example, burning occurs when someone exits a pool and redeems the underlying assets.

Collateralized currency. Paper currency backed by collateral such as gold, silver, or other assets.

Collateralized debt obligation. In traditional finance, this represents a debt instrument such as a mortgage. In *DeFI*, an example would be a *stablecoin* overcollateralized with a cryptoasset.

Consensus protocol. The mechanism whereby parties agree to add a new block to the existing *blockchain*. Both *Ethereum* and *bitcoin* use *proof of work*, but many other mechanisms exist, such as *proof of stake*.

Contract account. A type of account in *Ethereum* controlled by a *smart contract*.

Credit delegation. A feature whereby users can allocate collateral to potential borrowers who can use the collateral to borrow the desired asset.

Cryptocurrency. A digital token that is cryptographically secured and transferred using blockchain technology. Leading examples are *bitcoin* and *Ethereum*. Many different types of cryptocurrencies exist, such as *stablecoin* and tokens that represent digital and non-digital assets.

Cryptographic hash. A one-way function that uniquely represents the input data. It can be thought of as a unique digital fingerprint. The output is a fixed size even though the input can be arbitrarily large. A hash is not encryption because it does not allow recovery of the original

⁴⁶ See <https://www.creditslips.org/creditslips/2020/06/david-graebers-debt-the-first-5000-years.html>

message. A popular hashing algorithm is the SHA-256, which returns 256 bits or 64 *hexadecimal* characters. The *bitcoin blockchain* uses the SHA-256. *Ethereum* uses the Keccak-256.

DAO. See **Decentralized autonomous organization.**

dApp. A decentralized application that allows direct interactions between peers (i.e., removing the central clearing). These applications are permissionless and censorship resistant. Anyone can use them and no central organization controls them.

Decentralized autonomous organization (DAO). An algorithmic organization that has a set of rules encoded in a *smart contract* that stipulates who can execute what behavior or upgrade. A DAO commonly includes a *governance token*.

Decentralized exchange (DEX). A platform that facilitates token swaps in a noncustodial fashion. The two mechanisms for DEX liquidity are *order book matching* and *automated market maker*.

Decentralized finance (DeFi). A financial infrastructure that does not rely on a centralized institution such as a bank. Exchange, lending, borrowing, and trading are conducted on a peer-to-peer basis using *blockchain* technology and *smart contracts*.

Defi. See **Decentralized finance.**

Defi Legos. The idea that combining protocols to build a new protocol is possible. Sometimes referred to as DeFi Money Legos or composability.

DEX. See **Decentralized exchange.**

Digest. See **Cryptographic hash.** Also known as message digest.

Direct incentive. A payment or fee associated with a specific user action intended to be a reward for positive behavior. For example, suppose a *collateralized debt obligation* becomes undercollateralized. The condition does not automatically trigger liquidation. An *externally owned account* must trigger the liquidation, and a reward (direct incentive) is given for triggering the liquidation.

Double spend. A problem that plagued digital currency initiatives in the 1980s and 1990s: perfect copies can be made of a digital asset, so it can be spent multiple times. The *Satoshi Nakamoto* white paper in 2008 solved this problem using a combination of *blockchain* technology and *proof of work*.

Equity token. A type of cryptocurrency that represents ownership of an underlying asset or a pool of assets.

EOA. See **Externally owned account.**

ERC-20. Ethereum Request for Comments (ERC) related to defining the interface for fungible tokens. Fungible tokens are identical in utility and functionality. The US dollar is fungible currency in that all \$20 bills are identical in value and 20 \$1 bills are equal to the \$20 bill.

ERC-721. Ethereum Request for Comments (ERC) related to defining the interface for nonfungible tokens. Nonfungible tokens are unique and are often used for collectibles or specific assets, such as a loan.

ERC-1155. Ethereum Request for Comments (ERC) related to defining a multi-token model in which a contract can hold balances of a number of tokens, either fungible or non-fungible.

Ethereum. Second-largest cryptocurrency/*blockchain*, which has existed since 2015. The currency is known as ether (ETH). Ethereum has the ability to run computer programs known as *smart contracts*. Ethereum is considered a distributed computational platform.

Ethereum 2.0. A proposed improvement on the *Ethereum blockchain* that uses *horizontal scaling* and *proof-of-stake* consensus.

Externally owned account (EOA). An *Ethereum* account controlled by a specific user.

Fiat currency. Uncollateralized paper currency, which is essentially an IOU by a government.

Fintech (Financial Technology). A general term that refers to technological advances in finance. It broadly includes technologies in the payments, trading, borrowing, and lending spaces. Fintech often includes big data and machine learning applications.

Flash loan. An uncollateralized loan with zero counterparty risk and zero duration. A flash loan is used to facilitate arbitrage or to refinance a loan without pledging collateral. A flash loan has no counterparty risk because, in a single transaction, the loan is created, all buying and selling using the loan funding is completed, and the loan is paid in full.

Flash swap. Feature of some *DeFi* protocols whereby a contract sends tokens before the user pays for them with assets on the other side of the pair. A flash swap allows for near-instantaneous arbitrage. Whereas a *flash loan* must be repaid with the same asset, a flash swap allows the flexibility of repaying with a different asset. A key feature is that all trades occur within a single *Ethereum* transaction.

Fork. In the context of open source code, an upgrade or enhancement to an existing protocol that connects to the protocol's history. A user has the choice of using the old or the new protocol. If the new protocol is better and attracts sufficient mining power, it will win. Forking is a key mechanism to assure efficiency in *DeFi*.

Gas. A fee required to execute a transaction and to execute a *smart contract*. Gas is the mechanism that allows *Ethereum* to deal with the *halting problem*.

Geoblock. Technology that blocks users from certain countries bound by regulation that precludes the application.

Governance token. The right of an owner to vote on changes to the protocol. Examples include the MakerDAO MKR token and the Compound COMP token.

Halting problem. A computer program in an infinite loop. *Ethereum* solves this problem by requiring a fee for a certain amount of computing. If the *gas* is exhausted, the program stops.

Hash. See **Cryptographic hash**.

Hexadecimal. A counting system in base-16 that includes the first 10 numbers 0 through 9 plus the first six letters of the alphabet, a through f. Each hexadecimal character represents 4 bits, where 0 is 0000 and the 16th (f) is 1111.

Horizontal scaling. An approach that divides the work of the system into multiple pieces, retaining decentralization but increasing the throughput of the system through parallelization. This is also known as *sharding*. Ethereum 2.0 takes this approach in combination with a *proof-of-stake* consensus algorithm.

IDO. See **Initial DeFi Offering**.

Impermanent loss. Applies to *automated market makers (AMM)*, where a contract holds assets on both sides of a trading pair. Suppose the AMM imposes a fixed exchange ratio between the two assets, and both assets appreciate in market value. The first asset appreciates by more than the second asset. Users drain the first asset and the contract is left holding only the second asset. The impermanent loss is the value of the contract if no exchange took place (value of both tokens) minus the value of the contract after it was drained (value of second token).

Incentive. A broad term used to reward productive behavior. Examples include *direct incentives* and *staked incentives*.

Initial DeFi Offering (IDO). A method of setting an initial exchange rate for a new token. A user can be the first liquidity provider on a pair, such as, for example, the new token and a *stablecoin* such as USDC. Essentially, the user establishes an artificial floor for the price of the new token.

Invariant. The result of a constant product rule. For example, invariant = $S_A * S_B$, where S_A is the supply of asset A, and S_B is the supply of asset B. Suppose the instantaneous exchange rate is 1A:1B. The supply of asset A = 4 and the supply of asset B = 4. The invariant = 16. Suppose the investor wants to exchange some A for some B. The investor deposits 4 of A so that the contract has 8 A ($S_A = 4 + 4 = 8$). The investor can withdraw only 2 of asset B as defined by the invariant. The new supply of B is therefore 2 ($S_B = 4 - 2 = 2$). The invariant does not change, remaining at 16 = 2 * 8. The exchange rate does change, however, and is now 2A:1B.

Keeper. A class of *externally owned accounts* that is an incentive to perform an action in a *DeFi* protocol of a *dApp*. The keeper receives a reward in the form of a flat fee or a percentage of the incented action. For example, the keeper receives a fee for liquidating a *collateralized debt obligation* when it becomes undercollateralized.

KYC (Know Your Customer). A provision of US regulation common to financial services regulation requiring that users must identify themselves. This regulation has led to *geoblocking* of US customers from certain *decentralized exchange* functionalities.

Layer 2. A *scaling* solution built on top of a *blockchain* that uses cryptography and economic guarantees to maintain desired levels of security. For example, small transactions can occur using a multi-signature payment channel. The *blockchain* is only used when funds are added to the channel or withdrawn.

Liquidity provider (LP). A user that earns a return by depositing assets into a pool or a *smart contract*.

Mainnet. The fully-operational, production *blockchain* behind a token, such as the *Bitcoin* blockchain or the *Ethereum* blockchain. Often used to contrast with *testnet*.

Miner. Miners cycle through various values of a *nonce* to try to find a rare *cryptographic hash* value in a *proof-of-work blockchain*. A miner gathers candidate transactions for a new block, adds a piece of data called a *nonce*, and executes a *cryptographic hashing function*. The nonce is varied and the hashing continues. If the miner “wins” by finding a hash value that is very small, the miner receives a direct reward in newly minted cryptocurrency. A miner also earns an indirect reward, collecting fees for the transactions included in their block.

Miner extractable value. The profit derived by a miner. For example, the miner could front run a pending transaction they believe will increase the price of the cryptocurrency (e.g., a large buy).

Mint. An action that increases the supply of tokens and is the opposite of *burn*. Minting often occurs when a user enters a pool and acquires an ownership share. Minting and burning are essential parts of noncollateralized *stablecoin* models (i.e., when stablecoin gets too expensive more are minted, which increases supply and reduces prices). Minting is also a means to reward user behavior.

Networked liquidity. The idea that any exchange application can lever the liquidity and rates of any other exchange on the same *blockchain*.

Node. A computer on a network that has a full copy of a *blockchain*.

Nonce (Number Only Once). A counter mechanism for *miners* as they cycle through various values when trying to discover a rare *cryptographic hash* value.

Optimistic rollup. A scaling solution whereby transactions are aggregated off-chain into a single *digest* that is submitted to the chain on a periodic basis.

Oracle. A method whereby information is gathered outside of a *blockchain*. Parties must agree on the source of the information.

Order book matching. A process in which all parties must agree on the swap exchange rate. Market makers can post bids and asks to a *decentralized exchange (DEX)* and allow takers to fill the quotes at the pre-agreed price. Until the offer is taken, the market maker has the right to withdraw the offer or update the exchange rate.

Perpetual futures contract. Similar to a traditional futures contract, but without an expiration date.

Proof of stake. An alternative consensus mechanism, and a key feature of Ethereum 2.0, in which the staking of an asset on the next block replaces the mining of blocks as in *proof of work*. In *proof of work*, miners need to spend on electricity and equipment to win a block. In proof of stake, validators commit some capital (the stake) to attest that the block is valid. Validators make themselves available by staking their cryptocurrency and then they are randomly selected to propose a block. The proposed block needs to be attested by a majority of the other validators. Validators profit by both proposing a block as well as attesting to the validity of others' proposed blocks. If a validator acts maliciously, there is a penalty mechanism whereby their stake is *slashed*.

Proof of work (PoW). Originally advocated by Back in 2002, PoW is the consensus mechanism for the two leading *blockchains*: *Bitcoin* and *Ethereum*. Miners compete to find a rare *cryptographic hash*, which is hard to find but easy to verify. Miners are rewarded for finding the cryptographic hash and using it to add a block to the *blockchain*. The computing difficulty of finding the hash makes it impractical to go backward to rewrite the history of a leading blockchain.

Router contracts. In the context of *decentralized exchange*, a contract that determines the most efficient path of swaps in order to get the lowest slippage, if no direct trading pair is available e.g., on Uniswap.

Scaling risk. The limited ability of most current blockchains to handle a larger number of transactions per second. See *vertical scaling* and *horizontal scaling*.

Schelling-point oracle. A type of *oracle* that relies on the owners of a fixed supply of tokens to vote on the outcome of an event or report a price of an asset.

Sharding. A process of horizontally splitting a database, in our context, a blockchain. It is also known as *horizontal scaling*. This divides the work of the system into multiple pieces, retaining decentralization but increasing the throughput of the system through parallelization. *Ethereum 2.0* takes this approach with the goal of reducing network congestion and increasing the number of transactions per second..

Slashing. A mechanism in *proof of stake blockchain* protocols intended to discourage certain user misbehavior.

Slashing condition. The mechanism that triggers a *slashing*. An example of a slashing condition is when undercollateralization triggers a liquidation.

Smart contract. A contract activated when it receives ETH, or *gas*. Given the distributed nature of the *Ethereum blockchain*, the program runs on every *node*. A feature of the *Ethereum blockchain*, the main blockchain for *DeFi* applications.

Specie. Metallic currency such as gold or silver (or nickel and copper) that has value on its own (i.e., if melted and sold as a metal).

Stablecoin. A token tied to the value of an asset such as the US dollar. A stablecoin can be collateralized with physical assets (e.g., US dollar in USDC) or digital assets (e.g., DAI) or can be uncollateralized (e.g., AMPL and ESD).

Staking. The escrows of funds in a smart contract by a user who is subject to a penalty (*slashed* funds) if they deviate from expected behavior.

Staked incentive. A token balance custodied in a *smart contract* whose purpose is to influence user behavior. A staking reward is designed to encourage positive behavior by giving the user a bonus in their token balance based on the stake size. A staking penalty (*slashing*) is designed to discourage negative behavior by removing a portion of a user's token balance based on the stake size.

Swap. The exchange of one token for another. In *DeFi*, swaps are *atomic* and noncustodial. Funds can be custodied in a *smart contract* with withdrawal rights exercisable at any time before the swap is completed. If the swap is not completed, all parties retain their custodied funds.

Symmetric key cryptography. A type of cryptography in which a common key is used to encrypt and decrypt a message.

Testnet. An identically functioning *blockchain* to a *mainnet*, whose purpose is to test software. The tokens associated with the testnet when testing Ethereum, for example, are called test ETH. Test ETH are obtained for free from a smart contract that mints the test ETH (known as a faucet).

Transparency. The ability for anyone to see the code and all transactions sent to a *smart contract*. A commonly used blockchain explorer is etherscan.io.

Utility token. A fungible token required to utilize some functionality of a smart contract system or that has an intrinsic value defined by its respective smart contract system. For example, a *stablecoin*, whether collateralized or algorithmic, is a utility token.

Vampirism. An exact or near-exact copy of a *DeFi* platform designed to take liquidity away from an existing platform often by offering users *direct incentives*.

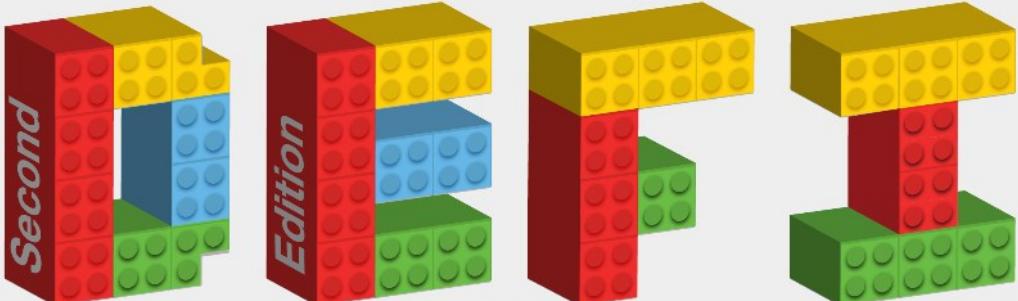
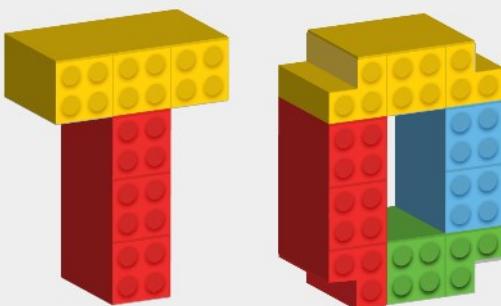
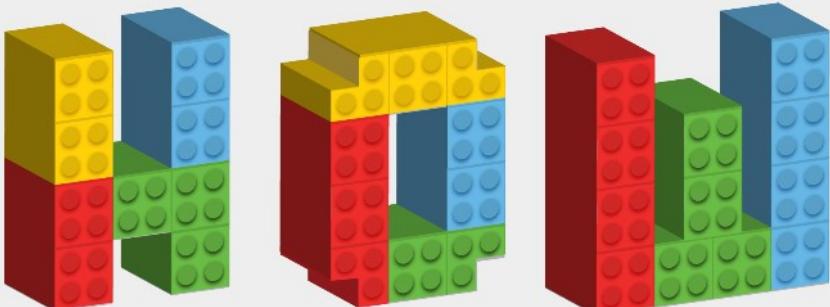
Vault. A smart contract that escrows collateral and keeps track of the value of the collateral.

Vertical scaling. The centralization of all transaction processing to a single large machine, which reduces the communication overhead (transaction/block latency) associated with a *proof-of-work blockchain*, such as *Ethereum*, but results in a centralized architecture in which one machine is responsible for a majority of the system's processing.

Yield farming. A means to provide contract-funded rewards to users for staking capital or using a protocol.

"This is an excellent resource for anyone who wants a comprehensive introduction to DeFi."

Kain Warwick, Founder of Synthetix



BEGINNER

Decentralized Finance is taking over the world.
Learn how to get started and join the revolution.



CoinGecko

How to DeFi: Beginner

2nd Edition, May 2021

Darren Lau, Daryl Lau, Teh Sze Jin, Kristian
Kho, Erina Azmi, Benjamin Hor, Lucius
Fang, Khor Win Win

Copyright © 2021 CoinGecko

1st edition, March 2020

2nd edition, May 2021

Layout: Anna Tan

teaspoonpublishing.com.my

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except brief extracts for the purpose of review, without the prior permission in writing of the publisher and copyright owners. It is also advisable to consult the publisher if in any doubt as to the legality of any copying which is to be undertaken.

“DeFi can be intimidating and overly complex, but this book makes it simple.”

– Seb Audet, Founder of Zapper

“If I didn't know anything about DeFi and needed to learn from scratch, this book is where I'd start.”

– Felix Feng, CEO of TokenSets

“This book makes it easy for beginners to get started with DeFi.”

– Hugh Karp, CEO of Nexus Mutual

“There is a lot of content about decentralized finance available but nothing matches this depth and comprehensiveness of this book.”

– Leighton Cusack, CEO of PoolTogether

“This is an excellent resource for anyone who wants a comprehensive introduction to DeFi.”

– Kain Warwick, Founder of Synthetix

“This book details the new economies created by a generation of bankless pioneers. It's the best introduction you could ask for.”

– Mariano Conti, Head of Smart Contracts at Maker Foundation

CONTENTS

<u>Introduction</u>	1
<u>Part One: Centralized & Decentralized Finance</u>	3
<u>Chapter 1: The Traditional Financial Institutions</u>	4
The Banks	4
1. Payment and Clearance System	5
2. Accessibility	6
3. Centralization & Transparency	7
Decentralized Finance vs. Traditional Finance	8
Recommended Readings	9
<u>Chapter 2: What is Decentralized Finance (DeFi)?</u>	10
The DeFi Ecosystem	11
How Decentralized is DeFi?	11
DeFi Key Categories	12
Stablecoins	12
Lending and Borrowing	13
Exchanges	13
Derivatives	13
Fund Management	14
Lottery	14
Payments	15
Insurance	15
Governance	16
Recommended Readings	16
<u>Part Two: Getting into DeFi</u>	18
<u>Chapter 3: The Decentralized Layer: Ethereum</u>	19
What is Ethereum?	19
What is a Smart Contract?	19
What is Ether (ETH)?	20
What is Gas?	21
What are Decentralized Applications (Dapps)?	21
What are the benefits of Dapps?	22
What are the disadvantages of Dapps?	22
What else can Ethereum be used for?	22
Ethereum's Future	23
Recommended Readings	24
<u>Chapter 4: Ethereum Wallets</u>	25
Custodial vs Non-Custodial Wallets	25
Which Wallet Should I Use?	26

<u>Mobile Users: Argent</u>	26
<u>Argent: Step-by-Step Guide</u>	27
<u>Desktop Users: Metamask</u>	33
<u>Metamask: Step-by-Step Guide</u>	34
<u>Recommended Readings</u>	39
Part Three: Deep Diving into DeFi	40
Chapter 5: Decentralized Stablecoins	41
<u>Maker</u>	43
<u>What is Maker?</u>	43
<u>What are the Differences between SAI and DAI?</u>	43
<u>How does Maker Govern the System?</u>	44
<u>Collateral Ratio</u>	44
<u>Stability Fee</u>	44
<u>DAI Savings Rate (DSR)</u>	45
<u>Motivations to Issue DAI</u>	45
<u>How do I get my hands on some Dai (DAI)?</u>	46
<u>Minting DAI</u>	46
<u>Trading DAI</u>	48
<u>Black Swan Event</u>	48
<u>Why use Maker?</u>	48
<u>Maker: Step-by-Step Guides</u>	50
<u>Minting your own DAI</u>	50
<u>Saving your DAI</u>	54
<u>Recommended Readings</u>	57
Chapter 6: Decentralized Lending and Borrowing	58
<u>Compound</u>	59
<u>How much interest will you receive, or pay?</u>	60
<u>Do I need to register for an account to start using Compound?</u>	61
<u>Compound Governance</u>	61
<u>How does Compound Governance work?</u>	62
<u>Start earning interest on Compound</u>	63
<u>What are cTokens?</u>	63
<u>Start borrowing on Compound</u>	64
<u>Price movement of collateral asset</u>	65
<u>Liquidation</u>	65
<u>Compound.Finance: Step-by-Step Guides</u>	66
<u>Supplying funds to the pool</u>	66
<u>Borrowing funds from the pool</u>	69
<u>Recommended Readings</u>	72
<u>Aave</u>	72
<u>How much interest will you receive or pay?</u>	74
<u>Which interest rate should I choose?</u>	75

<u>Do I need to register for an account to start using Aave?</u>	75
<u>Start earning interest on Aave</u>	75
<u>Start borrowing on Aave</u>	75
<u>Aave Governance and how it works</u>	76
<u>Aave: Step-by-Step Guide</u>	78
<u>Supplying funds from the pool</u>	78
<u>Borrowing funds from the pool</u>	81
<u>Recommended readings</u>	84
Chapter 7: Decentralized Exchanges (DEX)	85
<u>Types of DEX</u>	85
<u>Limitations of DEX</u>	86
<u>Uniswap</u>	87
<u>Liquidity Pools</u>	88
<u>Automated Market Maker Mechanism</u>	88
<u>How to get a token added on Uniswap?</u>	89
<u>Uniswap: Step-by-Step Guide</u>	91
<u>Swapping Tokens</u>	91
<u>Provide Liquidity</u>	94
<u>Withdraw Liquidity</u>	95
<u>Recommended Readings</u>	96
<u>DEX Aggregators</u>	97
<u>1inch</u>	97
<u>1inch: Step-by-Step Guide</u>	99
<u>Recommended Readings</u>	102
Chapter 8: Decentralized Derivatives	103
<u>Synthetix</u>	104
<u>What are Synthetic Assets (Synths)?</u>	104
<u>Why Synthetic Assets?</u>	105
<u>How are Synths Created?</u>	105
<u>What Assets do Synths Support?</u>	106
<u>Index Synths</u>	106
<u>sCEX</u>	107
<u>sDEFI</u>	107
<u>Synthetix Exchange</u>	107
<u>Synthetix: Step-by-Step Guide</u>	109
<u>Recommended Readings</u>	115
<u>Opyn</u>	116
<u>What is Opyn?</u>	116
<u>What are Options?</u>	116
<u>How does Opyn work?</u>	117
<u>How much do options cost?</u>	118
<u>Why would anyone sell protection on Opyn?</u>	118

<u>Being a Liquidity Provider on Uniswap</u>	119
<u>Selling oTokens on Uniswap</u>	119
<u>Is Opyn safe?</u>	119
<u>Opyn Version 2</u>	119
<u>Opyn Version 1: Step-by-Step Guide</u>	121
<u>Opyn Version 2: Step-by-Step Guide</u>	124
<u>Recommended Readings</u>	126
<u>Chapter 9: Decentralized Fund Management</u>	127
<u>TokenSets</u>	128
<u>What kinds of Sets are there?</u>	128
<u>Index Sets</u>	128
<u>Yield Farming Sets</u>	129
<u>How are Sets helpful?</u>	130
<u>TokenSets: Step-by-Step Guide</u>	131
<u>Recommended Readings</u>	136
<u>Chapter 10: Decentralized Lottery</u>	137
<u>PoolTogether</u>	138
<u>What is PoolTogether?</u>	138
<u>Why bother with Decentralized Lotteries?</u>	139
<u>What's the Catch?</u>	139
<u>What are the odds of winning?</u>	139
<u>What's new in PoolTogether's Version 3?</u>	139
<u>PoolTogether Token</u>	140
<u>Pool Together Governance</u>	141
<u>PoolTogether: Step-by-Step Guide</u>	142
<u>Recommended Readings</u>	145
<u>Chapter 11: Decentralized Payments</u>	146
<u>Sablier</u>	146
<u>What is Sablier?</u>	146
<u>What Does Streaming Payment Mean?</u>	147
<u>Why is this important?</u>	147
<u> Trust</u>	147
<u> Timing</u>	147
<u>Sablier Step-by-Step Guide</u>	149
<u>Conclusion</u>	152
<u>Recommended Readings</u>	153
<u>Chapter 12: Decentralized Insurance</u>	154
<u>Nexus Mutual</u>	155
<u>What is Nexus Mutual?</u>	155
<u>What event is covered by Nexus Mutual?</u>	156
<u>How does coverage work?</u>	157

<u>How is the coverage priced?</u>	157
<u>How to purchase cover?</u>	158
<u>NXM Token</u>	158
<u>wNXM Token</u>	158
<u>What is a Risk Assessor?</u>	159
<u>Has NXM ever paid out claims before?</u>	159
<u>Nexus Mutual: Step-by-Step Guide</u>	159
<u>Armor</u>	163
<u> Armor: Step-by-Step Guide</u>	164
<u>Other Insurance Platforms</u>	166
<u> Nsure Network</u>	167
<u> Cover Protocol</u>	167
<u> Conclusion</u>	168
<u> Recommended Readings</u>	169
<u>Chapter 13: Governance</u>	170
<u>Aragon</u>	171
<u> What is Aragon?</u>	171
<u> What is the Aragon Court?</u>	172
<u> How to become a juror?</u>	172
<u> Who uses Aragon?</u>	173
<u> Aragon Step-by-Step Guide</u>	173
<u> Recommended Readings</u>	176
<u>Snapshot</u>	176
<u> What is Snapshot?</u>	176
<u> What are the shortcomings of using Snapshot?</u>	176
<u> Snapshot: Step-by-Step Guide</u>	177
<u> Recommended Readings</u>	182
<u>Chapter 14: DeFi Dashboard</u>	183
<u>What is a Dashboard?</u>	183
<u>Zapper: Step-by-Step Guide</u>	184
<u>Part Four: DeFi in Action</u>	187
<u>Chapter 15: DeFi in Action</u>	188
<u>Surviving Argentina's High Inflation</u>	188
<u>Uniswap Ban</u>	191
<u>Chapter 16: DeFi is the Future, and the Future is Now</u>	194
<u>What about DeFi User Experience?</u>	196
<u>Closing Remarks</u>	198
<u>Appendix</u>	199
<u>CoinGecko's Recommended DeFi Resources</u>	199

<u>Information</u>	199
<u>News Sites</u>	199
<u>Newsletters</u>	199
<u>Podcast</u>	200
<u>Youtube</u>	200
<u>Bankless Level-Up Guide</u>	200
<u>Projects We Like Too</u>	200
<u>Dashboard Interfaces</u>	200
<u>Decentralized Exchanges</u>	200
<u>Exchange Aggregators</u>	201
<u>Lending and Borrowing</u>	201
<u>Prediction Markets</u>	201
<u>Taxes</u>	201
<u>Wallet</u>	201
<u>Yield Optimisers</u>	201
<u>References</u>	202
<u>Chapter 1: Traditional Financial Institutions</u>	202
<u>Chapter 2: What is Decentralized Finance (DeFi)?</u>	203
<u>Chapter 3: The Decentralized Layer: Ethereum</u>	203
<u>Chapter 4: Ethereum Wallets</u>	203
<u>Chapter 5: Decentralized Stablecoins</u>	204
<u>Chapter 6: Decentralized Borrowing and Lending</u>	205
<u>Chapter 7: Decentralized Exchange (DEX)</u>	205
<u>Chapter 8: Decentralized Derivatives</u>	206
<u>Chapter 9: Decentralized Fund Management</u>	206
<u>Chapter 10: Decentralized Lottery</u>	207
<u>Chapter 11: Decentralized Payment</u>	208
<u>Chapter 12: Decentralized Insurance</u>	208
<u>Chapter 13: Governance</u>	210
<u>Chapter 14: DeFi Dashboard</u>	210
<u>Chapter 15: DeFi in Action</u>	210
<u>Glossary</u>	211

INTRODUCTION

So much development has taken place in DeFi since we published the First Edition of *How to DeFi: Beginner* in March 2020! DeFi, the acronym for **Decentralized Finance**, is currently one of the fastest-growing sectors in the blockchain and cryptocurrency space. With increasing institutional acceptance, one may soon see DeFi integrated within the traditional financial system in the future.

Starting from humble beginnings in late-2017, DeFi gained traction in the wider community during the summer of 2020 and has not stopped since; protocols in this space are constantly innovating. In this Second Edition of *How to DeFi: Beginner* book, we will help you delve into the world of DeFi by categorizing the industry into concise chapters, now updated with the latest development in the space (if you read the First Edition of *How to DeFi: Beginner*, some of the content might already be outdated—that's how fast it moves!)

As previously done, we will be explaining what DeFi is, how it is crucial for the community, and the various elements of DeFi, such as decentralized stablecoins, exchanges, lending, derivatives, and insurance. In each of these chapters, we will provide step-by-step guides to assist you in interacting with at least one of the DeFi protocols.

Throughout the book, we will have **Recommended Readings** at the end of each chapter. In these sections, we will share supplementary reading materials that we believe will be useful as you dive deeper into the DeFi

How to DeFi: Beginner

ecosystem. All credits, of course, go to their respective authors. Kudos to them for making DeFi more accessible!

How to DeFi: Beginner book is aimed at DeFi beginners. For the DeFi enthusiasts who want to delve deeper, we have published the *How to DeFi: Advanced* book with more in-depth analysis. Therefore, we recommend beginners start with this book before proceeding to the Advanced version.

We hope that the contents of this book will help you get up to speed with DeFi. We look forward to having you join us in this movement!

CoinGecko Research Team

Darren Lau, Daryl Lau, Teh Sze Jin, Kristian Kho, Erina Azmi, Benjamin Hor, Lucius Fang, Khor Win Win

1 May 2021

PART ONE: CENTRALIZED & DECENTRALIZED FINANCE

CHAPTER 1: THE TRADITIONAL FINANCIAL INSTITUTIONS

In our attempt to shed light on people new to DeFi, we will start by first going through the basics of how traditional financial institutions work. For simplicity, we will focus on the highest leveraged institutions in the traditional financial system, the banks, and discuss its key areas to see the potential risks.

The Banks

Banks are the financial industry's giants that facilitate payments, accept deposits, and offer lines of credit to individuals, businesses, other financial institutions, and even governments. They are so large that the total market capitalization of the top 10 banks in the world is \$2 trillion. In April 2021, the total market capitalization of the entire cryptocurrency market surpassed \$2 trillion.

Banks are vital parts of the moving machine that is the financial industry—they enable money to move around the world by providing value transfer services (deposit, withdrawal, transfers), extending credit lines (loans), and more. However, banks are managed by humans and governed by policies that are prone to human-related risks such as mismanagement and corruption.

Top 10 Global Banks 2019			
Rank	Bank	Country	Market Cap. (\$ bn)
1	ICBC	China	338
2	China Construction Bank	China	287
3	Agricultural Bank of China	China	243
4	Bank of China	China	230
5	JP Morgan Chase	US	209
6	Bank of America	US	189
7	Wells Fargo	US	168
8	Citigroup	US	158
9	HSBC	UK	147
10	Mitsubishi UFJ	Japan	146

Source: [Top 1000 World Banks 2019](#)

The global financial crisis of 2008 exemplified excessive risk-taking by banks, and governments were forced to make massive bailouts of the banks. The crisis exposed the shortcomings of the traditional financial system and highlighted a need for it to be better.

DeFi seeks to build a better financial landscape made possible by the advent of the internet and blockchain technology, particularly in three key segments of the banking system:

1. Payment & clearance system (remittance)
2. Accessibility
3. Centralization & Transparency

1. Payment and Clearance System

If you have tried to send money to someone or a business in another country, you know this pain all too well—remittances involving banks worldwide typically take a few working days to complete and involve all sorts of fees. To make matters worse, there may also be issues with documentation, compliance with anti-money laundering laws, privacy concerns, and more.

For example, suppose you live in the US and would like to send \$1,000 from your bank account in the US to your friend's bank account in Australia. There are typically three fees involved: the exchange rate from your bank, the international wire outbound fee, and the international wire inbound fee. Additionally, it will take a few working days for the recipient to receive the money, depending on the recipient bank's location.

Cryptocurrencies that power the DeFi movement allow you to bypass intermediaries who take the lion's share of these transfers' profits. It is likely to be quicker as well—your transfers will be processed with no questions asked with relatively lower fees compared to banks. For example, the transfer of cryptocurrencies to any account in the world would take anywhere between 15 seconds to 5 minutes depending on several factors, along with a small fee.

2. Accessibility

Chances are if you are reading this book, you are banked and have access to financial services offered by banks—to open a savings account, take a loan, make investments, and more. However, many others are less fortunate and do not have access to even the most basic savings account.



Heatmap of the Unbanked (Source: [Global Findex, World Bank, 2017](#))

The World Bank estimates that as of 2017, there are 1.7 billion people who do not own an account at a financial institution, and more than half of them are from developing nations. They come mainly from poor households, and some of their main reasons for not having a bank account are poverty, geographical, and trust issues.

However, the World Bank also estimates that two-thirds of the 1.7 billion unbanked population have access to mobile phones. Therefore, with an internet connection, DeFi Dapps can be the gateway for millions of people in the unbanked population to access financial products and conduct financial transactions without going through lengthy verification processes as required by traditional banks.

DeFi represents a movement that seeks to push borderless, censorship-free, and accessible financial products for all. DeFi protocols do not discriminate and level the playing field for everyone.

3. Centralization & Transparency

There is no denying that traditional, regulated financial institutions that comply with government laws and regulations are some of the most secure places to park funds. But they are not without flaws—even large banks can fail. Washington Mutual, with over \$188 billion in deposits, and Lehman Brothers, with \$639 billion in assets, have both failed in 2008. In the US alone, over 500 bank failures have been recorded.

Banks are one of the centralized points of failure in the financial system. The fall of Lehman Brothers triggered the start of the 2008 financial crisis. The centralization of power and funds in the hands of the banks is dangerous, and rightfully so, looking at past incidents.

Transparency also ties into this—there is no way for regular investors to know what financial institutions do entirely. Some of the events leading up to the 2008 financial crisis included credit rating agencies giving AAA ratings (best & safest investments) to high-risk mortgage-backed securities.

It will be different with DeFi. DeFi protocols built on top of public blockchains such as Ethereum are mostly open-sourced for audit and transparency purposes. They usually have decentralized governing organizations to ensure that everyone knows what is happening and that no bad actors can single-handedly make bad decisions.

DeFi protocols are written as lines of codes—you cannot cheat the codes as it treats every participant equally without discrimination. The codes run exactly as they are programmed to, and any flaws quickly become evident as it is open for public scrutiny.

However, this means that only people who can understand code would be able to determine the actual functionalities of the end-product. In practice, most users are unable to read the code and rely on other factors such as the developers' reputation, word-of-mouth, other developers' comments, and community approval. In other words, DeFi protocols rely on decentralized peer-to-peer review. DeFi's biggest strength is thus its ability to remove intermediaries while operating with zero censorship.

Decentralized Finance vs. Traditional Finance

Friction, inaccessibility, and regulatory uncertainties are some of the major issues plaguing the current banking system. Unfortunately, not everyone is privileged to be banked in the current financial system, nor can they compete financially on a level playing field.

One only needs to look at the conundrum behind the GameStop (GME) trades, where a bunch of small-time investors decided to buy GME shares because big-time hedge funds heavily shorted it. Financial apps like Robinhood had to step in and restrict GME trades because of “extraordinary volatility”. Some might take this at face value, while others will note that one of Robinhood’s biggest customers is Citadel LLC. Citadel is a company that invested in Melvin Capital, which lost billions of dollars over their short on GME.

The DeFi movement is about bridging these gaps and making finance accessible to everyone without any form of censorship. In short, DeFi opens up huge windows of opportunities and allows users to access various financial instruments without any restriction on race, religion, age, nationality, or geography.

When comparing both traditional and decentralized financial products, there will be pros and cons on each side. In this book, we will walk you through the concepts and possibilities of decentralized finance so that you will know how to use its best features to solve real-world problems.

In [Chapter 2](#) we will provide an overview of DeFi and some of its Decentralized Applications to help capture the underlying notions on how DeFi works.

Recommended Readings

1. Decentralized Finance vs Traditional Finance: What You Need To Know (Stably) <https://medium.com/stably-blog/decentralized-finance-vs-traditional-finance-what-you-need-to-know-3b57aed7a0c2>
2. The 7 Major Flaws of the Global Financial System (Jeff Desjardins) <https://www.visualcapitalist.com/7-major-flaws-global-financial-system>
3. Decentralized Finance: An Emerging Alternative to the Global Financial System (Frank Cardona) <https://www.visualcapitalist.com/decentralized-finance/>
4. How Decentralized Finance Could Make Investing More Accessible (Jeff Desjardins) <https://www.visualcapitalist.com/how-decentralized-finance-could-make-investing-more-accessible/>
5. What is Decentralized Finance? <https://101blockchains.com/decentralized-finance-defi/>

CHAPTER 2: WHAT IS DECENTRALIZED FINANCE (DEFI)?

Decentralized Finance or DeFi is the movement that allows users to utilize financial services such as borrowing, lending, and trading without the need to rely on centralized entities. These financial services are provided via Decentralized Applications (Dapps), the majority of which are deployed on the Ethereum platform.

While it is helpful to understand how Ethereum works to visualize the ecosystem better, you do not need to be an Ethereum expert to utilize the tools offered by DeFi. We will touch more on Ethereum in the next chapter.

DeFi is not a single product or company, but a range of financial services which emulates traditional financial industries, including banking, insurance, bonds, money markets and more. DeFi Dapps enable users to combine these services to achieve desired financial goals. It is often called money LEGOs due to its composability.

For DeFi Dapps to work, a collateral locked into smart contracts is usually required. The cumulative collateral locked in DeFi Dapps is often referred to as the Total Value Locked, which serves as a growth indicator of the DeFi ecosystem. In our earlier edition, we highlighted that the Total Value Locked at the start of 2019 measured around \$275 million and reached a high of \$1.2 billion in February 2020.

As of April 2021, the Total Value Locked is a staggering \$67 billion in Ethereum alone which goes to show how far we have come. This does not even include other blockchain networks such as Binance, Solana, and others. The Total Value Locked for all the chains collectively adds up to \$86 billion.¹

The DeFi Ecosystem

With such rapid growth, it would be impossible for us to cover everything DeFi offers in this book. That is why we have selected a few categories and DeFi Dapps that we believe are important and crucial for beginners to understand before stepping into the DeFi ecosystem.

These DeFi Dapps stand to revolutionize traditional financial services by removing the need for any middlemen. However, you should note that DeFi in its current state is still highly nascent and experimental, with many projects rapidly improving each day. As time goes on, DeFi may develop further and look entirely unrecognizable from what it is today. Nevertheless, it is helpful to understand the early beginnings of DeFi, and you can still take advantage of the features offered by DeFi Dapps today with the right know-how.

How Decentralized is DeFi?

It is not easy to answer how decentralized DeFi is. For simplicity, we will separate the degrees of decentralization into three categories: centralized, semi-decentralized and completely decentralized.

1. Centralized
 - Characteristics: Custodial, uses centralized price feeds, centrally-determined interest rates, centrally-provided liquidity for margin calls
 - Examples: Salt, BlockFi, Nexo and Celsius
2. Semi-Decentralized (has one or more of these characteristics but not all)
 - Characteristics: Non-custodial, decentralized price feeds, permissionless initiation of margin calls, permissionless margin

¹ DeFi Llama. Retrieved April 1, 2021 from <https://defillama.com/home>

- liquidity, decentralized interest rate determination, decentralized platform development/updates
 - Examples: Compound, MakerDAO, dYdX, and bZx
- 3. Completely Decentralized
 - Characteristics: Every component is decentralized
 - Examples: No DeFi protocol is completely decentralized yet.

Currently, most DeFi dapps are sitting in the semi-decentralized category. You may read more on the various decentralization components in [Kyle Kistner's article](#) in the Recommended Readings. Now that you have a better understanding of what being decentralized means, let's move on to the key categories of DeFi.

DeFi Key Categories

In this book, we will be covering nine major categories of DeFi. Although Governance is not strictly a DeFi category, we believe it is also important to discuss how protocols govern themselves, thus it deserves a chapter of its own.

1. Stablecoins

The prices of cryptocurrencies are known to be highly volatile. It is common for cryptocurrencies to have intraday swings of over 10%. To mitigate this volatility, stablecoins that are pegged to other stable assets such as the USD were created.

Tether (USDT) was one of the first centralized stablecoins to be introduced. Every USDT is supposedly backed by \$1 in the issuer's bank account. However, one major downside to USDT is that users need to trust that the USD reserves are fully collateralized and actually exist.

Decentralized stablecoins aim to solve this trust issue. They are created via an over-collateralization method, operate fully on decentralized ledgers, and are governed by decentralized autonomous organizations. Anyone can publicly audit their reserves.

What is Decentralized Finance (DeFi)

While stablecoins are not financial applications themselves, they are essential in making DeFi applications more accessible to everyone by having a stable store of value.

2. Lending and Borrowing

Traditional financial systems require users to have bank accounts to utilize their services, a luxury that 1.7 billion people currently do not have. Borrowing from banks comes with other restrictions, such as having a good credit score and having sufficient collateral to convince the banks that one is credit-worthy and able to repay a loan.

Decentralized lending and borrowing remove this barrier, allowing anyone to collateralize their digital assets and use this to obtain loans. One can also earn a yield on their assets and participate in the lending market by contributing to lending pools and earning interest on these assets. With decentralized lending and borrowing, there is no need for a bank account nor checking for credit-worthiness.

3. Exchanges

To exchange one cryptocurrency for another, one can use exchanges such as Coinbase or Binance. Exchanges like these are centralized exchanges, meaning they are both the intermediaries and custodians of the traded assets. Users of these exchanges do not have complete control of their assets, putting their assets at risk if the exchanges get hacked and are unable to repay their obligations.

Decentralized exchanges aim to solve this issue by allowing users to exchange cryptocurrencies without giving up custody of their coins. By not storing any funds on centralized exchanges, users do not need to trust the exchanges to stay solvent.

4. Derivatives

A derivative is a contract whose value is derived from another underlying asset such as stocks, commodities, currencies, indexes, bonds, or interest rates.

Traders can use derivatives to hedge their positions and decrease their risk in any particular trade. For example, imagine you are a glove manufacturer and want to hedge yourself from an unexpected increase in rubber price. You can buy a futures contract from your supplier to deliver a specific amount of rubber at a specific future delivery date at an agreed price today.

Derivatives contracts are mainly traded on centralized platforms. DeFi platforms are starting to build decentralized derivatives markets. We will go through this in further detail in [Chapter 8](#).

5. Fund Management

Fund management is the process of overseeing your assets and managing its cash flow to generate a return on your investments. There are two main types of fund management - active and passive fund management. Active fund management has a management team making investment decisions to beat a particular benchmark, such as the S&P 500. Passive fund management does not have a management team but is designed in such a way to mimic the performance of a particular benchmark as closely as possible.

In DeFi, some projects have started to allow passive fund management to occur in a decentralized manner. The transparency of DeFi makes it easy for users to track how their funds are being managed and understand the cost they will be paying.

6. Lottery

As DeFi continues to evolve, creative and disruptive financial applications will emerge, democratizing accessibility and removing intermediaries. Putting a DeFi spin onto lotteries allows for the removal of custodianship of the pooled capital into a smart contract on the Ethereum Blockchain.

With the modularity of DeFi, it is possible to link a simple lottery Dapp to another DeFi Dapp and create more value. One DeFi Dapp that we will explore in this book allows participants to pool their capital together. The pooled money is then invested into a DeFi lending Dapp

and the interest earned is given to a random winner at a set interval. Once the winner is selected, the lottery purchasers get their lottery tickets refunded, ensuring no-loss to all participants.

7. Payments

A key role of cryptocurrency is to allow decentralized and trustless value transfer between two parties. With the growth of DeFi, more creative payment methods are being innovated and experimented upon.

One such DeFi project explored in this book aims to change the way we approach payment by reconfiguring payments as streams instead of transactions we are familiar with. The possibility of providing payments as streams open up a plethora of potential applications of money. Imagine “pay-as-you-use” but on a much more granular scale and higher accuracy.

The nascent of DeFi and the rate of innovation will undoubtedly introduce new ways of thinking on how payments work to address many of the current financial system’s shortcomings.

8. Insurance

Insurance is a risk management strategy in which an individual receives financial protection or reimbursement against losses from an insurance company in the event of an unfortunate incident. It is common for individuals to purchase insurance on cars, home, health, and life. But is there decentralized insurance for DeFi?

All of the tokens locked within smart contracts are potentially vulnerable to smart contract exploits due to the large potential payout possible. While most projects have gotten their codebases audited, we never know if the smart contracts are truly safe, and there is always a possibility of a hack that may result in a loss. The risks highlight the need for purchasing insurance, especially if one deals with large amounts of funds on DeFi. We will explore several decentralized insurance options in this book.

9. Governance

Governance is essentially crypto's idea of business management. In order for DeFi protocols to manage a project, governance tokens are often introduced to give users voting power and have a say in the protocol's roadmap. Naturally, multiple toolkits and Dapps have also been developed to facilitate effective governance and complement existing systems.

Recommended Readings

1. Decentralized Finance Explained (Yos Riady)
<https://yos.io/2019/12/08/decentralized-finance-explained/>
2. A beginner's guide to DeFi (Linda J. Xie)
<https://nakamoto.com/beginners-guide-to-defi/>
3. A Beginner's Guide to Decentralized Finance (DeFi) (Coinbase)
<https://blog.coinbase.com/a-beginners-guide-to-decentralized-finance-defi-574c68ff43c4>
4. The Complete Beginner's Guide to Decentralized Finance (DeFi) (Binance)
<https://www.binance.vision/blockchain/the-complete-beginners-guide-to-decentralized-finance-defi>
5. 2019 Was The Year of DeFi (and Why 2020 Will be Too) (Mason Nystrom)
<https://consensys.net/blog/news/2019-was-the-year-of-defi-and-why-2020-will-be-too/>
6. DeFi: What It Is and Isn't (Part 1) (Justine Humenansky)
<https://medium.com/coinmonks/defi-what-it-is-and-isnt-part-1-f7d7e7afee16>
7. How Decentralized is DeFi? A Framework for Classifying Lending Protocols (Kyle Kistner)
<https://hackernoon.com/how-decentralized-is-defi-a-framework-for-classifying-lending-protocols-90981f2c007f>
8. How Decentralized is “Decentralized Finance”? (Aaron Hay)
<https://medium.com/coinmonks/how-decentralized-is-decentralized-finance-89aea3070e8f>
9. Mapping Decentralized Finance
<https://outlierventures.io/wp-content/uploads/2019/06/Mapping-Decentralised-Finance-DeFi-report.pdf>

What is Decentralized Finance (DeFi)

10. Market Report: 2019 DeFi Year in Review
<https://defirate.com/market-report-2019/>
11. DeFi #3 – 2020: The Borderless State of DeFi
<https://research.binance.com/analysis/2020-borderless-state-of-defi>
12. Decentralized Finance with Tom Schmidt (Software Engineering Daily)
<https://softwareengineeringdaily.com/2020/02/25/decentralized-finance-with-tom-schmidt/>

PART TWO: GETTING INTO DEFI

CHAPTER 3: THE DECENTRALIZED LAYER: ETHEREUM

What is Ethereum?

As mentioned in [Chapter 1](#), the majority of the DeFi Dapps are currently being built on the Ethereum blockchain. But what exactly is Ethereum? Ethereum is a global, open-source platform for decentralized applications. You can think of Ethereum as a world computer that no one can shut down. On Ethereum, software developers can write smart contracts that control digital value through a set of criteria and are accessible anywhere in the world.

In this book specifically, we will be exploring Decentralized Applications (Dapps) that provide financial services known as DeFi. Smart contracts that software programmers write are the building blocks of these Dapps. These smart contracts are then deployed to the Ethereum network, where they will run 24/7. The network will maintain the digital value and keep track of the latest state.

What is a Smart Contract?

A smart contract is a programmable contract that allows two counterparties to set conditions of a transaction without needing to trust another third party for the execution.

For example, if Alice wants to set up a trust fund to pay Bob \$100 at the start of each month for the next 12 months, she can program a smart contract to:

1. Check the current date
2. At the start of each month, send Bob \$100 automatically
3. Repeat until the fund in the smart contract is exhausted

Using a smart contract, Alice has bypassed the need to have a trusted third-party intermediary (lawyers, escrow agents, etc.) to send the trust fund to Bob and made the process transparent to all involved parties.

Smart contracts work on the “if this, then that” principle. Whenever a specific condition is fulfilled, the smart contract will carry out the operation as programmed.

Multiple smart contracts are combined to operate with each other, known as decentralized applications (Dapps), to fulfill more complex processes and computations.

What is Ether (ETH)?

Ether is the native currency of the Ethereum blockchain.

It is like money and can be used for everyday transactions similar to Bitcoin. You can send Ether to another person to purchase goods and services based on the current market value. The Ethereum blockchain records the transfer and ensures the finality of the transaction.

Besides that, Ether is also used to pay the fee that allows smart contracts and Dapps to run on the Ethereum network. You can think of executing smart contracts on the Ethereum network as driving a car. To drive a car, you require fuel. To execute a smart contract on Ethereum, you need to use Ether to pay a fee known as Gas.

Ether is slowly evolving to become its unique reserve currency and store of value. Currently, in the DeFi ecosystem, Ether is the preferred asset choice used as the collateral underlying many DeFi Dapps. It provides safety and transparency to this financial system. Don’t worry if this confuses you, as we will be covering the topic in further depth throughout this book.

What is Gas?

On Ethereum, all transactions and smart contract executions require a small fee to be paid. This fee is called Gas. In technical terms, Gas refers to the unit of measure on the amount of computational effort required to execute an operation or a smart contract. The more complex the execution operation is, the more Gas that is needed to fulfill that operation. Gas fees are paid entirely in ETH.

The price of Gas can fluctuate from time to time depending on the network demand. When more people interact on the Ethereum blockchain, such as transacting in ETH or executing smart contract transactions, due to the limited amount of computing resources on the network, Gas price can increase. Conversely, when the network is underutilized, the Gas price would decrease.

Users may set gas fees manually. When the network gets congested due to high utilization, miners will prioritize transactions with the highest gas fees. Validated transactions will be finalized and added to the blockchain. If gas fees paid are too low, the transactions will be queued, taking a while to complete. Therefore, transactions with lower-than-average gas fees can take much longer to complete.

Gas price is typically denoted in *gwei*.

$1 \text{ gwei} = 0.000000001 \text{ ether}$

Assume a smart contract execution to transfer tokens require 21,000 gas units.

Assume the average market rate for gas price is 3 *gwei*.

$21,000 \text{ gas} \times 3 \text{ gwei} = 63,000 \text{ gwei} = 0.000063 \text{ ETH}$

When executing the transactions, you will pay a gas fee of 0.000063 ETH to process and validate your transaction in the network.

Example of how gas fees are calculated

What are Decentralized Applications (Dapps)?

In the context of Ethereum, Dapps are interfaces that interact with the blockchain through the use of smart contracts. Dapps look and behave like

regular web and mobile applications, except that they interact with a blockchain and in different ways. Some of the ways include requiring ETH to use the Dapp, storing user data onto the blockchain such that it is immutable, and so on.

What are the benefits of Dapps?

Dapps are built on top of decentralized blockchain networks such as Ethereum and usually have the following benefits:

- **Immutability:** Nobody can change any information once it's on the blockchain.
- **Tamper-proof:** Smart contracts published onto the blockchain cannot be tampered with without alerting every other participant on the blockchain.
- **Transparent:** Smart contracts powering Dapps are openly auditable.
- **Availability:** As long as the Ethereum network remains active, Dapps built on it will remain active and usable.

What are the disadvantages of Dapps?

While a blockchain offers many benefits, there are also many different downsides:

- **Immutability:** Smart contracts are written by humans and can only be as good as the person who wrote them. As human errors are unavoidable, immutable smart contracts have the potential to compound mistakes into big problems.
- **Transparent:** Openly auditable smart contracts can also become attack vectors for hackers as the hackers can view the code to find exploits.
- **Scalability:** In most cases, the bandwidth of a Dapp is limited to the blockchain it resides on.

What else can Ethereum be used for?

Besides creating Dapps, Ethereum can also be used for two other functions—creating Decentralized Autonomous Organizations (DAO) or issuing other cryptocurrencies.

A DAO is a fully autonomous organization that is not governed by a single person but is instead governed through code. This code is based on smart contracts and enables DAOs to replace how traditional organizations are typically run. As it runs on code, it would be protected from human intervention and will operate transparently. There would be no effect by any outside influence. Governance decisions or rulings would be decided via DAO token voting.

Speaking of tokens, Ethereum can be used as a platform to create other cryptocurrencies. There are currently two popular protocols for tokens on the Ethereum Network: ERC-20 and ERC-721. Both ERC-20 and ERC-721 are protocol standards that define rules and standards for issuing tokens on Ethereum.

ERC-20 tokens are fungible, meaning they are interchangeable and of the same value. On the other hand, ERC-721 tokens are non-fungible, meaning they are unique and non-interchangeable. A simple analogy would be to think of ERC-20 as money and ERC-721 as collectibles like action figures or baseball cards

Ethereum's Future

Ethereum's popularity continues to grow as it becomes the central pillar of DeFi growth. With the first mover's advantage in hand, the number of users and transactions continues to grow each day. While many herald this as a success for DeFi, the surge in demand is putting a major strain on the network.

Rising gas fees are among the most significant issues as users now need to pay exorbitant fees during peak hours. The high gas fees have led to the rapid development of competing blockchains such as Polkadot, where the network is touted to be more efficient (alternative blockchains are discussed more in-depth in our Advanced Book).

To ensure the continued success of the Ethereum network, the Ethereum community is planning to introduce an upgrade known as ETH 2.0. ETH 2.0 is a massive undertaking that spans over three years and utilizes 'sharding'

techniques. Once the update is fully incorporated, the network will become more scalable, solving the high gas fees issue.

And that's it for Ethereum! If you are keen to own your first cryptocurrency or try your first Dapp, we will be covering several interesting DeFi protocols in the following chapters. We will be providing overviews and step-by-step guides. Before you can begin your journey, you will first need an Ethereum Wallet.

Recommended Readings

1. What is Ethereum? [The Most Updated Step-by-Step-Guide!] (Ameer Rosic) <https://blockgeeks.com/guides/ethereum/>
2. Smart Contracts: The Blockchain Technology That Will Replace Lawyers (Ameer Rosic) <https://blockgeeks.com/guides/smart-contracts/>
3. What is Ethereum Gas? [The Most Comprehensive Step-By-Step Guide Ever!] (Ameer Rosic) <https://blockgeeks.com/guides/ethereum-gas/>
4. The trillion-dollar case for ETH (Lucas Campbell) <https://bankless.substack.com/p/the-trillion-dollar-case-for-eth-eb6>
5. Ethereum: The Digital Finance Stack (David Hoffman) <https://medium.com/pov-crypto/ethereum-the-digital-finance-stack-4ba988c6c14b>
6. Ether: A New Model for Money (David Hoffman) <https://medium.com/pov-crypto/ether-a-new-model-for-money-17365b5535ba>
7. The Eth2 vision <https://ethereum.org/en/eth2/vision/>

CHAPTER 4: ETHEREUM WALLETS

A wallet is a user-friendly interface to the blockchain network. It manages your private keys, which are basically keys to the lock on your cryptocurrencies' vault. Wallets allow you to receive, store and send cryptocurrencies.

Custodial vs Non-Custodial Wallets

There are two kinds of wallets—custodial and non-custodial wallets. Custodial wallets are wallets where third-parties keep and maintain control over your cryptocurrencies on your behalf. Non-custodial wallets are wallets where you take full control and ownership of your cryptocurrencies. This is similar to the mantra espoused by many people in the blockchain industry to “be your own bank”.

By using a custodial wallet, you trust an external party to store your coins safely. This may be convenient as you do not need to worry about private key security and only worry about account credentials security, similar to how you would have to protect your email account. However, by trusting a third party with your cryptocurrencies, you open yourself up to the risk of the custodian losing your cryptocurrencies through mismanagement or hacks. There have been numerous incidents where custodial wallets lost their cryptocurrencies. The most prominent example was Mt. Gox, which lost over 850,000 bitcoin worth over \$450 million in 2014.

By using a non-custodial wallet, you trust no external party and only yourself to ensure that your cryptocurrencies are safe. However, by using a non-custodial wallet, you pass the burden of security to yourself, and you have to be fully equipped to store your private keys safely. If you lose your private keys, you will lose access to your cryptocurrencies too.

At CoinGecko, we believe in the “not your keys, not your coin” mantra. We believe that you should educate yourself in all the best security practices and trust only yourself to keep your coins safe.

Which Wallet Should I Use?

There are many cryptocurrency wallets out there in the market. In this book, we will walk through two DeFi friendly wallets for you to easily start interacting with the Ethereum network.

Mobile Users: Argent

For mobile users, you may consider using the Argent wallet. Argent is a non-custodial wallet that offers ease-of-use and high security, something which does not always go hand-in-hand. It does so by utilizing Argent Guardians, which are people, devices, or third-party services that can verify your identity.

Examples include family and friends who are also Argent users, other hardware or Metamask wallets, or two-factor authentication services. By utilizing this limited circle of trust network, Argent is rethinking the need for paper-based seed phrase backups when recovering accounts.

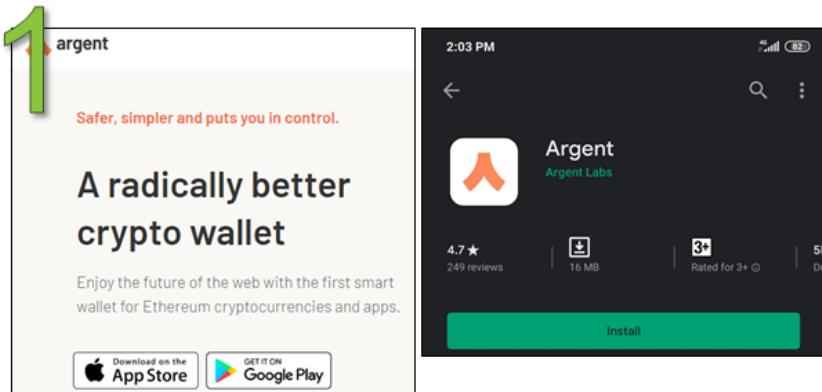
Argent Guardians allows you to lock your wallet and instantly freeze all funds if you believe your wallet has been compromised. Your wallet will be automatically unlocked after five days, or you can request for your Argent Guardian to unlock it sooner.

You may also set additional security measures to improve your wallet security, such as a daily transaction limit. This is useful in preventing hackers from siphoning funds from your Argent wallet if they gain access to your wallet. Whenever your daily transaction limit is hit, you will receive a

notification, and any transactions over the limit will be delayed for over 24 hours. You can, of course, authorize legitimate large transactions over the limit through the help of your Argent Guardians.

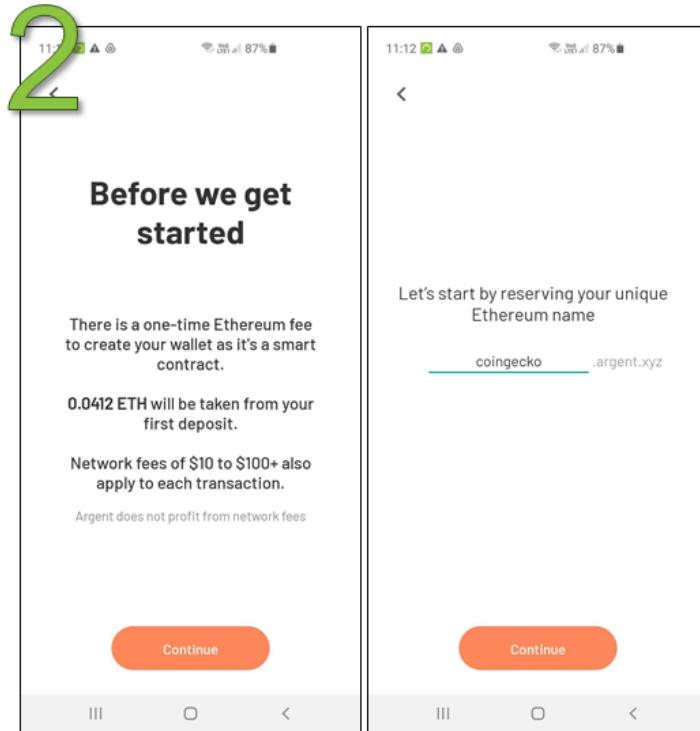
Argent requires a one-time Ethereum fee to create your wallet on the network. Network fees also apply to each transaction made through the app (Argent does not profit from the network fees). With Argent wallet, you can easily interact with DeFi Dapps directly from the wallet without the need to use another app or device.

Argent: Step-by-Step Guide



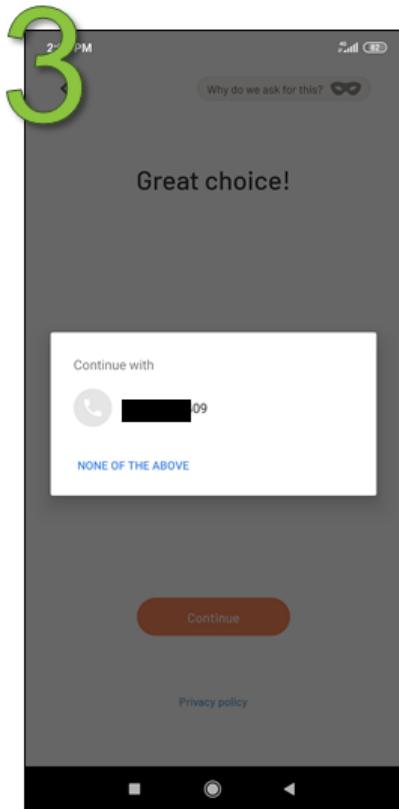
Step 1

- Go to <https://argent.link/coingecko>
- Download the app on your mobile phone



Step 2

- Once downloaded, Argent will inform you of Ethereum network charges that will be imposed later
- Choose a unique Ethereum name for your argent wallet



Step 3

- Argent will ask if you want to add your phone number for added security and verification purposes

4



And which email address should we use to notify you of important activity?

 **argent**

Please verify your email

Press the button below to verify your email. This link will expire in 15 minutes.

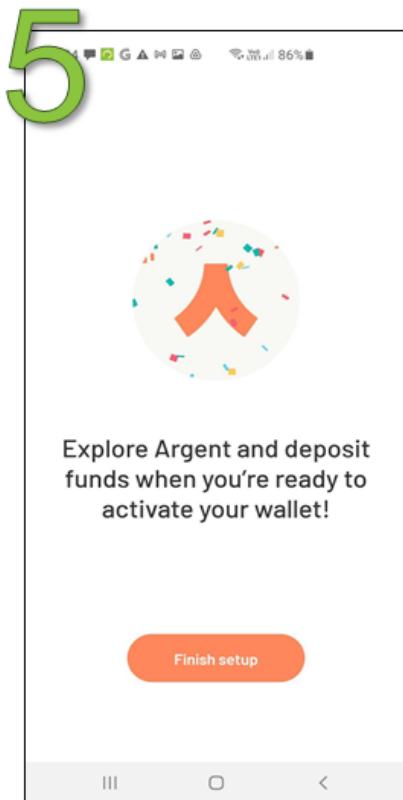
[Verify email](#)

[Verify Email](#)

[Privacy policy](#)

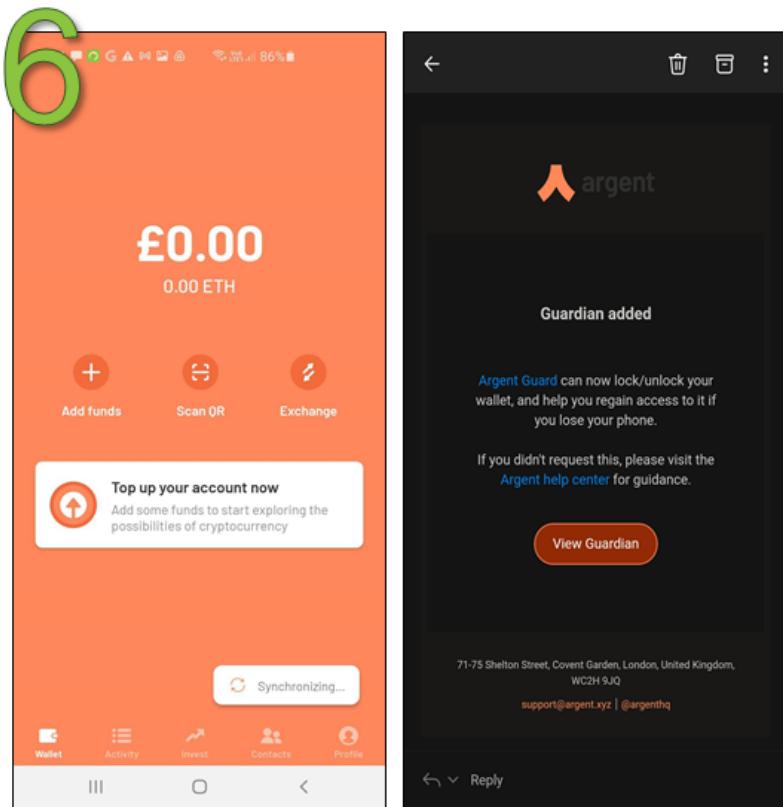
Step 4

- Afterward, Argent will ask for your email for verification purposes.



Step 5

- You will get an email notification once your wallet is ready to use!



Step 6

- You start depositing or sending cryptocurrencies to other people. Do consider adding additional Argent Guardians to improve your security.

Desktop Users: Metamask

For desktop users, you may use Metamask, a web browser extension available on Chrome, Firefox, Opera, and Brave browsers. Like Argent, Metamask is a non-custodial wallet and acts as both a wallet and an interaction bridge for the Ethereum network. It should be noted that Metamask also has an app for mobile phones. However, the app may have difficulties interacting with Dapps as some Dapps are not optimized for mobile devices.

You can store your Ethereum and ERC20 tokens on Metamask. Acting as an interaction bridge, Metamask enables you to use all Decentralized Applications (Dapps) hosted on the Ethereum Network.

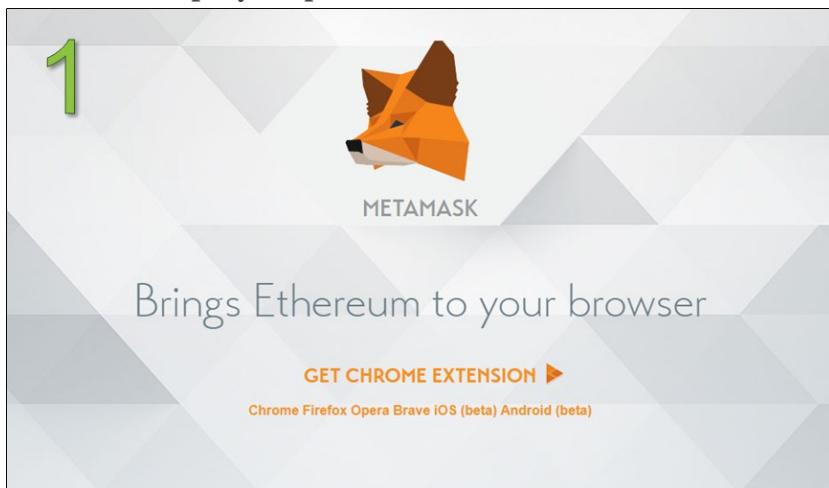
Without the use of an interaction bridge like MetaMask, your browser would not be able to access the Ethereum blockchain unless you were running a full Ethereum node and had the entire Ethereum blockchain of over 400GB downloaded on your computer. On a technical level, MetaMask does this by injecting a javascript library known as web3.js written by the core Ethereum developers into your browser's page to enable you to interact with the Ethereum network easily.

Metamask makes interaction with DeFi Dapps on the Ethereum network very convenient on your laptop or PC. They are secured to some degree as it requires you to sign each interaction and transaction you execute on the network. Metamask also has an in-built token swap feature, which allows you to exchange other tokens directly from the wallet.

However, it would be best if you took precautions to keep your Metamask safe and secure. Anybody who has your password or seed phrase (a secret phrase given to you during wallet sign-up) will have complete control of your wallet.

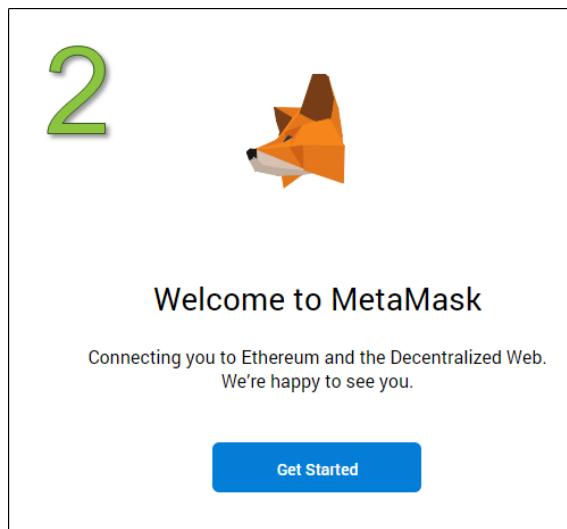
Most DeFi Dapps can be accessed using Metamask, and in the later chapters, you will notice that the step-by-step guides have been completed using Metamask.

Metamask: Step-by-Step Guide



Step 1

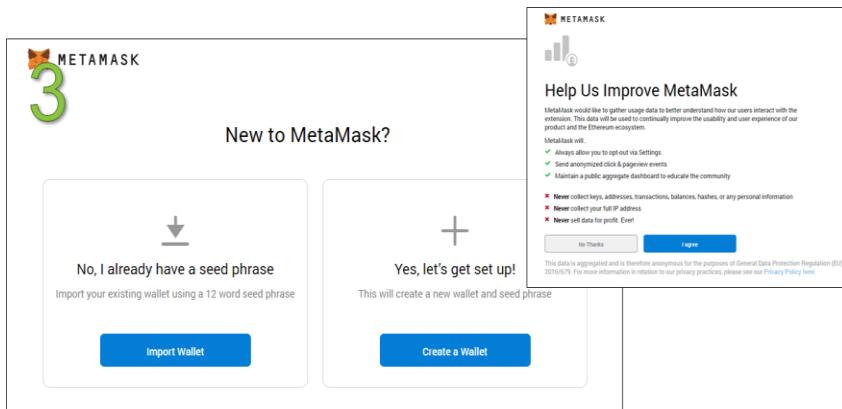
- Go to <https://metamask.io/>
- Download extension for the browser of your choice



Step 2

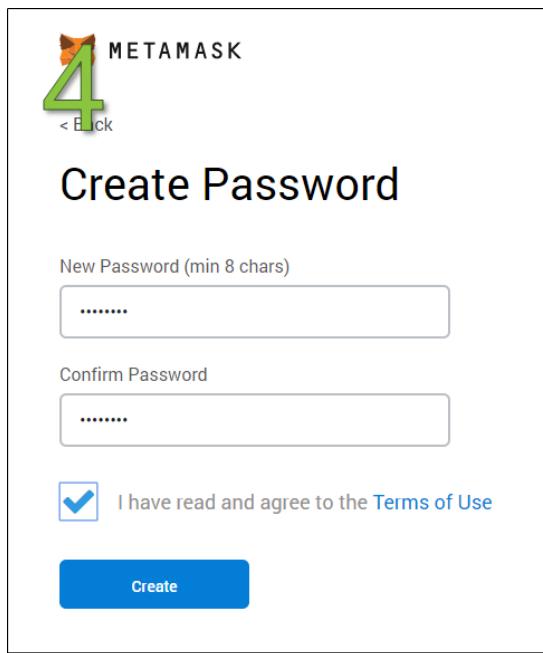
- After you have downloaded the extension, click "Get Started"

Ethereum Wallets



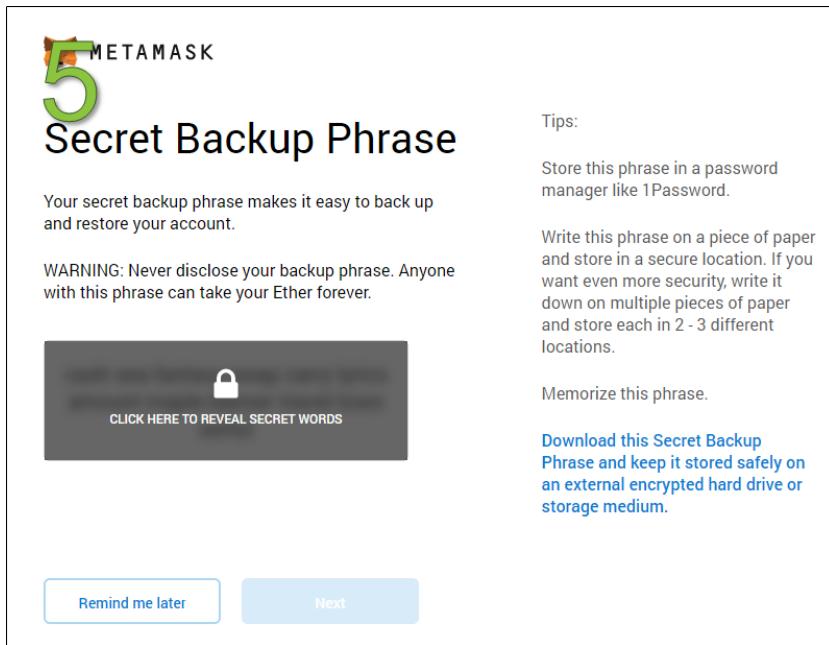
Step 3

- Click “Create a Wallet” and click “Next”.



Step 4

- Create your password.



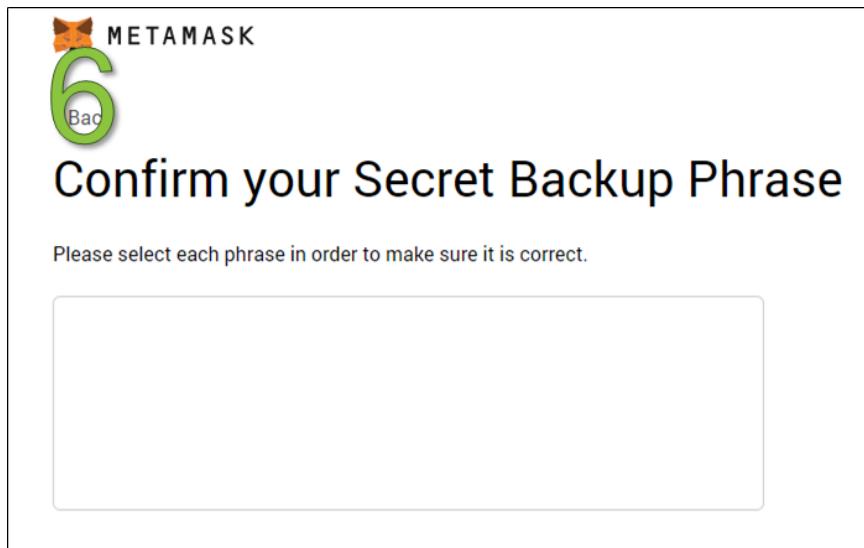
The screenshot shows the MetaMask wallet interface for generating a secret backup phrase. At the top left is the Metamask logo. Below it, the text "Secret Backup Phrase" is displayed in large, bold letters. A subtext explains that the secret backup phrase makes it easy to back up and restore the account. A warning message states: "WARNING: Never disclose your backup phrase. Anyone with this phrase can take your Ether forever." Below this is a large button with a lock icon and the text "CLICK HERE TO REVEAL SECRET WORDS". At the bottom of the screen are two buttons: "Remind me later" and "Next". To the right of the screen, there is a section titled "Tips:" with three items:

- Store this phrase in a password manager like 1Password.
- Write this phrase on a piece of paper and store in a secure location. If you want even more security, write it down on multiple pieces of paper and store each in 2 - 3 different locations.
- Memorize this phrase.

Below these tips is a blue link: "Download this Secret Backup Phrase and keep it stored safely on an external encrypted hard drive or storage medium."

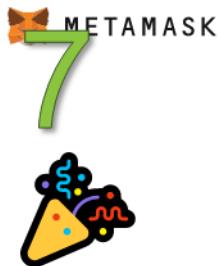
Step 5 (IMPORTANT! READ CAREFULLY!)

- You will be given a Secret Backup Phrase
- NEVER lose it
- NEVER show it to anyone
- If you lose the phrase, you can't retrieve it
- If anyone else has it, they are able to access your wallet and do anything with it



Step 6

- You will be prompted to write the given secret backup phrase to confirm that you have noted it down



Congratulations

You passed the test - keep your seedphrase safe, it's your responsibility!

Tips on storing it safely

- Save a backup in multiple places.
- Never share the phrase with anyone.
- Be careful of phishing! MetaMask will never spontaneously ask for your seed phrase.
- If you need to back up your seed phrase again, you can find it in Settings -> Security.
- If you ever have questions or see something fishy, email support@metamask.io.

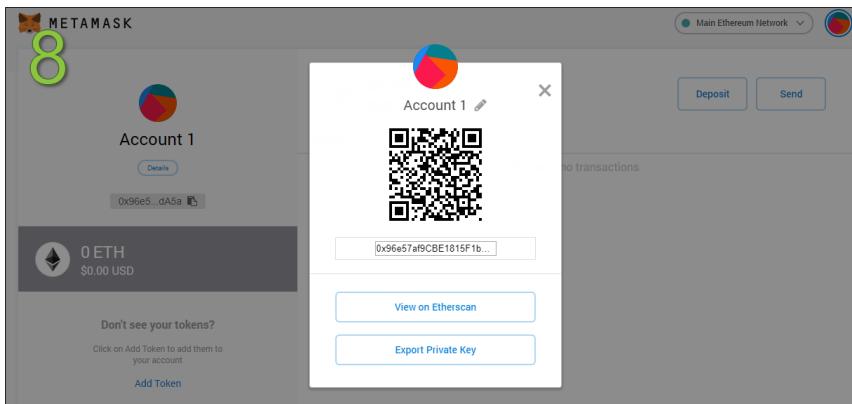
*MetaMask cannot recover your seedphrase. [Learn more.](#)

All Done

Step 7

- Congratulations! Your wallet is now created! You can use it to store Ethereum and ERC20 tokens

Ethereum Wallets



Step 8

- Below is your public key or your Ethereum address to your wallet
- Your QR code can be scanned if anyone wants to send you coins.

Recommended Readings

1. Argent: The quick start guide (Matthew Wright)
<https://medium.com/argenthq/argent-the-quick-start-guide-13541ce2b1fb>
2. A new era for crypto security (Itamar Lesuisse)
<https://medium.com/argenthq/a-new-era-for-crypto-security-57909a095ae3>
3. A Complete Beginner's Guide to Using MetaMask (Ian Lee)
<https://www.coingecko.com/buzz/complete-beginners-guide-to-metamask>
4. MyCrypto's Security Guide For Dummies And Smart People Too (Taylor Monahan) <https://medium.com/mycrypto/mycryptos-security-guide-for-dummies-and-smart-people-too-ab178299c82e>

PART THREE: DEEP DIVING INTO DEFI

CHAPTER 5: DECENTRALIZED STABLECOINS

The prices of cryptocurrencies are extremely volatile. In order to mitigate this volatility, stablecoins pegged to other stableassets such as the USD were created. Stablecoins help users hedge against this price volatility and allow for a reliable medium of exchange. Stablecoins have since quickly evolved to be a vital component of DeFi that is pivotal to this modular ecosystem.

There are 49 stablecoins currently listed on [CoinGecko](#). The top 5 stablecoins have a market capitalization totaling over \$59.8 billion.

Top 5 Cryptocurrency Stablecoins (1 April 2021)		
Rank	Bank	Market Cap. (\$ billion)
1	Tether (USDT)	40.8
2	USD Coin (USDC)	10.8
3	Binance USD (BUSD)	3.5
4	DAI (TUSD)	3.0
5	TerraUSD (USDT)	1.6

Source: CoinGecko.com

We will be looking into USD-pegged stablecoins in this chapter. Not all stablecoins are the same as they employ different mechanisms to keep their peg against USD. There are three types of stablecoins—fiat-collateralized, crypto-collateralized, and algorithmic stablecoins. Most stablecoins use the fiat-collateralized system to maintain their USD peg.

For simplicity, we will look at two USD stablecoins, Tether ([USDT](#)) and Dai ([DAI](#)) to showcase the differences in their pegging management. We will not be covering algorithmic stablecoins, a more recent DeFi innovation, in this book. If you are interested to learn more about algorithmic stablecoins, you can refer to our *How to DeFi: Advanced* book.

Tether ([USDT](#)) pegs itself to \$1 by maintaining reserves of \$1 per Tether token minted. While Tether is the largest and most widely used USD stablecoin with daily trading volumes averaging approximately \$113 billion in March 2021, Tether reserves are kept in financial institutions with little oversight, thus users will have to trust Tether as an entity to actually have the reserve amounts that they claim. Tether is therefore a **centralized, fiat-collateralized stablecoin**.

On the other hand, DAI ([DAI](#)) is collateralized using cryptocurrencies such as Ethereum ([ETH](#)). Its value is pegged to \$1 through protocols voted on by a decentralized autonomous organization and smart contracts. At any given time, users can easily validate that the collateral used to generate DAI exists. DAI is thus a **decentralized, crypto-collateralized stablecoin**.

Based on the top 5 stablecoins' market capitalization, Tether dominates the stablecoin market with approximately 68% of the market share. Although DAI's market share only stands at about 5%, its trading volume has been increasing at a much faster rate. DAI's trading volume increased by over 158% relative to Tether's growth of 95% throughout Q1 2021.

DAI is the native stablecoin used most widely in the DeFi ecosystem. It is one of the preferred USD stablecoin used in DeFi trading, lending, and more. To understand DAI further, we will introduce you to its platform, Maker.

Maker



What is Maker?

Maker is a smart-contract platform that runs on the Ethereum blockchain and **has two tokens**: DAI (both algorithmically pegged to \$1) and its governance token—Maker ([MKR](#)).

Dai (DAI) was launched in November 2019 and is also known as Multi-Collateral DAI. It is currently backed by a basket of third-party assets such as Ether (ETH), Basic Attention Token (BAT), USDC and Wrapped Bitcoin (wBTC). More types of collateral are continuously added based on community proposals.

Maker (MKR) is Maker's governance token, and users can use it to vote for improvements on the Maker platform via the Maker Improvement Proposals. Maker is a type of organization known as a Decentralized Autonomous Organization (DAO). We will look further into this under the governance subsection.

What are the Differences between SAI and DAI?

Maker initially started out on [19 December 2017](#) with the Single Collateral DAI. It was minted using Ether ([ETH](#)) as the sole collateral. On [18 November 2019](#), Maker announced the launch of the new Multi-Collateral DAI, which can be minted using either Ether ([ETH](#)) and/or Basic Attention Token ([BAT](#)) as collateral.

In March 2020, Maker introduced its first centralized-back collateral USDC to help manage its liquidity crisis and DAI price instability during the Black Thursday crash. Currently, users are able to submit proposals at [MakerDAO's forum](#) where the community will evaluate and decide on onboarding new collateral types.

To reiterate,

$$\text{Single-Collateral DAI} = \text{Legacy DAI} = \text{SAI}$$

$$\text{Multi-Collateral DAI} = \text{New DAI} = \text{DAI}$$

SAI has already since been phased out and Multi-Collateral DAI is currently the de-facto stablecoin standard maintained by Maker.

How does Maker Govern the System?

Recall our brief mention of Decentralized Autonomous Organization (DAO)? That's where the Maker (MKR) token comes in - MKR holders have voting rights proportional to the amount of MKR tokens they own in the DAO and can vote on parameters governing the Maker Protocol.

The parameters that the MKR holders vote on are vital in keeping the ecosystem healthy, which in turn helps ensure that DAI remains pegged to \$1. We will briefly go through three key parameters which you will need to know in the DAI stablecoin ecosystem:

I. Collateral Ratio

The amount of DAI that can be minted is dependent on the collateral ratio.

For example:

$$\text{Wrapped Bitcoin (wBTC) collateral ratio} = 150\%$$

$$\text{Basic Attention Token (BAT) collateral ratio} = 150\%$$

Therefore, a collateral ratio of 150% means that in order to mint \$100, you need to deposit a minimum of \$150 worth of wBTC or BAT.

II. Stability Fee

It is equivalent to the ‘interest rate’ which you are required to pay along with the principal debt of the vault. The stability fee for each Vault type changes as a result of the decisions of MKR token holders who govern

the protocol. These decisions are based on the recommendation of Maker's internal risk teams who perform risk assessments on collaterals used in the system. For example, the current [stability fee for BAT is 6.0%](#) as of April 2021.

III. DAI Savings Rate (DSR)

The DAI Savings Rate (DSR) is the interest earned by holding DAI over time. It also acts as a monetary tool to influence the demand of DAI. The [DSR rate is set at 0.01%](#) as of April 2021.

Motivations to Issue DAI:

Why would you want to lock up a higher value of collateral such as ETH only to issue DAI with a lower value? You could have sold your assets directly to USD instead.

There are three possible cases:

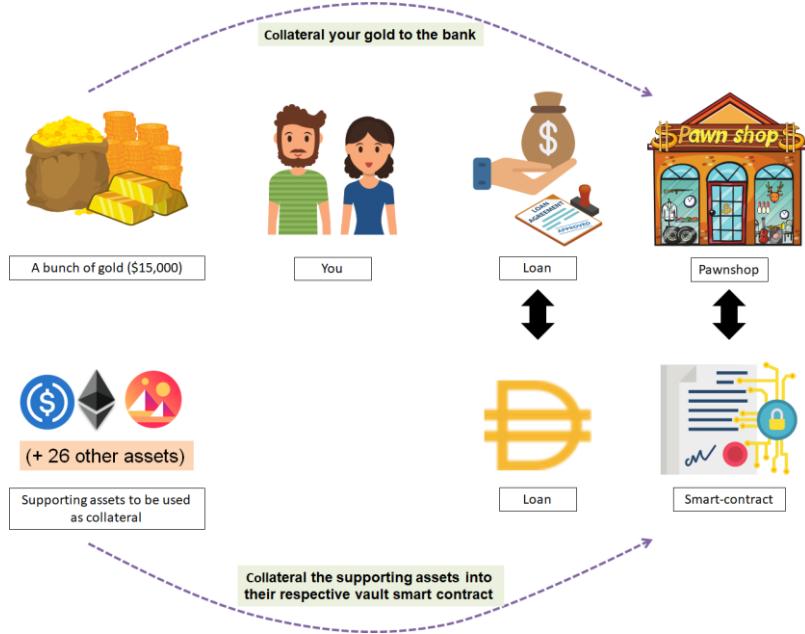
- I. You need cash now and you have an asset that you believe will be worth more in the future.
 - In this case, you could hold your asset in the Maker vault and get the money now by issuing DAI.
- II. You need cash now but do not want to risk triggering a taxable event when selling your asset.
 - Instead, you will draw the loans by issuing DAI.
- III. Investment Leverage
 - You are able to conduct investment leverage on your assets given that you believe the value of your assets would go up.

How do I get my hands on some Dai (DAI)?

There are two ways you can get your hands on some Dai (DAI):

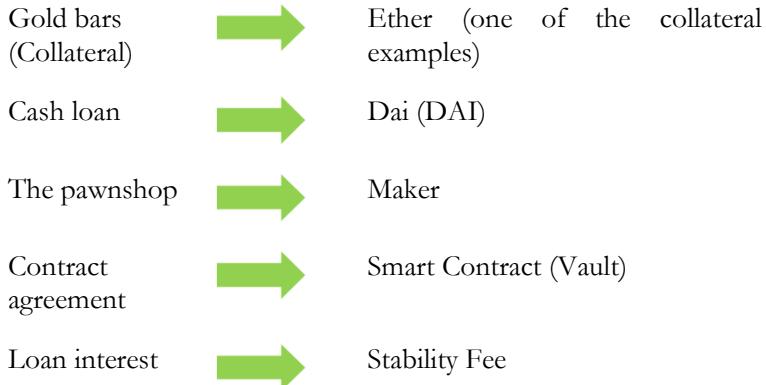
1. Minting DAI

We will walk through how DAI can be minted using a pawnshop analogy.



Let's assume that one day you are in need of \$10,000 cash, but all you have are gold bars worth \$15,000 at home. Believing that the price of gold will increase in the future, instead of selling the gold bars for cash, you decide to go to a pawnshop to borrow \$10,000 cash by putting your gold bars as collateral for it. The pawnshop agrees to lend you \$10,000 with an interest of 8% for the cash loan. Both of you sign a contract agreement to finalize the transaction.

Now let's change the terminology to get the narrative of DAI:



What happens is that you will mint or ‘borrow’ DAI via the Maker platform by putting your Ether (ETH) as collateral. You will have to repay your ‘loan’ along with the ‘loan interest’ which is the stability fee when you want to redeem your ETH at the end of your loan.

To provide an overview, let’s walk through how you can mint your own DAI.

On the Maker platform (www.oasis.app), you can borrow DAI by putting your ETH into the vault. Assuming ETH is currently worth \$150, you can thus lock 1 ETH into the vault and receive a maximum of 100 DAI (\$100) with a 150% collateral ratio. There are currently three types of vaults for ETH with different collateral ratios, but for the sake of simplicity, we will assume that the collateral ratio is 150%, which is the ETH-A vault.

You should not draw out the maximum of 100 DAI that you are allowed to but leave some buffer in the event that ETH price decreases. It is advisable to give a wider gap to ensure your collateral ratio always remains above 150%. This ensures that your vault will not be liquidated and charged the 13% liquidation penalty in the event that ETH falls in price and your collateral ratio falls below 150%.

2. Trading DAI

The above methods are all the ways DAI is created. Once DAI is created, you can send it anywhere you want. Some users may send their DAI to cryptocurrency exchanges. You may also buy DAI from these secondary markets without the need to mint them.

Buying DAI this way is easier as you do not need to lock up collateral and do not have to worry about the collateral ratio and stability fee.

We will keep this section brief - you can check out CoinGecko for the [list of exchanges](#) that trades DAI.

Black Swan Event



A black swan event is an unpredictable and extreme event that may cause severe consequences. In the case where Maker's collateral has a significant drop in price, an Emergency Shutdown will be triggered. It is a process used as a last resort to settle the Maker Platform by shutting the system down. The process is to ensure the holders of DAI holders and Vault users receive the net value of assets they are entitled to.

In March 2020 (also known as Black Thursday), an Emergency Shutdown was almost triggered, where the price of ETH dropped by 50% within 24 hours. Maker mitigated the impact through its automated debt auction (despite its late trigger due to the sharp drop) and quickly introduced USDC as a new collateral type to back the system.

Why use Maker?

As previously mentioned in Section 2: Stablecoins, there are many stablecoins, and the core distinctions of these stablecoins lie in their protocol. Unlike most stablecoin platforms, Maker is fully operating on the distributed ledger. Thus, Maker inherently possesses the characteristics of the blockchain: secured, immutable, and most importantly, transparent.

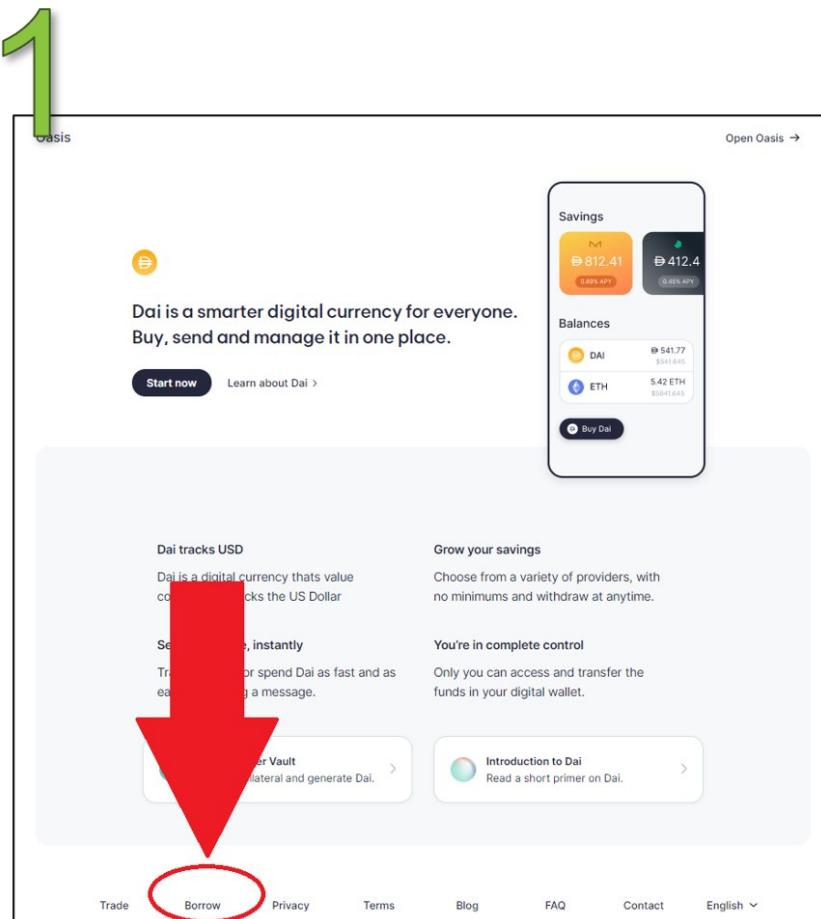
Decentralized Stablecoins

Additionally, Maker's infrastructures have strengthened the system's security with comprehensive risk protocols and mechanisms via real-time information.

And that's it for Makers' Stablecoin, DAI. If you're keen to get started or test it out, we've included step-by-step guides on how to (i) mint some DAI for yourself and (ii) save DAI to earn interest. Otherwise, head on to the next section to read more on the next DeFi app!

Maker: Step-by-Step Guides

Minting your own DAI



The screenshot shows the homepage of the Oasis app. At the top left is a large green number '1'. At the top right is a link to 'Open Oasis →'. The main content area features a yellow circular icon with a dollar sign. Below it, text reads: 'Dai is a smarter digital currency for everyone. Buy, send and manage it in one place.' There are two buttons: 'Start now' and 'Learn about Dai >'. To the right is a sidebar with 'Savings' and 'Balances' sections, and a 'Buy Dai' button. Below the sidebar are two columns of text: 'Dai tracks USD' and 'Grow your savings', followed by 'Send Dai, instantly' and 'You're in complete control'. A large red arrow points down from the top towards the 'Borrow' button. The bottom navigation bar includes links for Trade, Borrow (which is circled in red), Privacy, Terms, Blog, FAQ, Contact, and English dropdown.

Step 1

- Go to <https://oasis.app/>
- Click Borrow
- You will be asked to connect your wallet. Connecting your wallet is free, all you need to do is sign a transaction

2

SELECT COLLATERAL VAULT MANAGEMENT GENERATE DAI CONFIRMATION

Select a collateral type

Each collateral type has its own risk parameters. You can lock up additional collateral types later.

COLLATERAL TYPE	STABILITY FEE	LIQ RATIO	LIQ FEE	YOUR BALANCE
<input checked="" type="radio"/> ETH-A	8.000 %	150 %	13.00 %	0.106 ETH
<input type="radio"/> BAT-A	8.000 %	150 %	13.00 %	0 BAT

Choose which coins you want to collateralize to borrow Dai.

Note: Only one type of coins at a time to borrow Dai.

Back Continue

Stability Fee
The fee calculated based on the outstanding debt of your Vault. This is continuously added to your existing debt.

Liquidation Ratio
The collateral-to-dai ratio at which the Vault becomes vulnerable to liquidation.

Liquidation Fee
The fee that is added to the total outstanding DAI debt when a liquidation occurs.

Step 2

- Click “Start Borrow” on the borrow page (<https://oasis.app/borrow/>)
- Choose which cryptoasset you want to lock-in (collateralize)

3

Deposit ETH and Generate Dai

Different collateral types have different risk parameters and collateralization ratios.

How much ETH would you like to lock in your Vault?

The amount of ETH you lock up determines how much Dai you can generate.

0.2 ETH

YOUR BALANCE 0.203 ETH

How much Dai would you like to generate?

Generate an amount that is safely above the liquidation ratio.

21 DAI

MAX AVAIL TO GENERATE 32.5 DAI

Max amount of Dai you could generate

Your Collateralization Ratio 232.20% (Min 150%)

Your Liquidation Price \$157.50

Current ETH Price \$243.81

Stability Fee 8.000%

Note: Minimum amount you could generate is 20 DAI

Back Continue

Step 3

- In this example, we chose to lock-in ETH
- Insert any amount you wish to lock. Note: The minimum amount of borrowing is 20 DAI
- Click Continue and follow the instructions provided

4

ETH-A Vault #4807

Liquidation price <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> 195.19 USD (ETH/USD) Current price information (ETH/USD) 243.81 USD Liquidation penalty 13.00% </div>	Collateralization ratio <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> 187.36 % Minimum ratio 150.00% Stability fee 8.0000% </div>															
ETH locked <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> ETH locked 0.2000 ETH 48.78 USD Deposit Able to withdraw 0.0399 ETH 9.72 USD Withdraw </div>	Outstanding Dai debt <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> Outstanding Dai debt 26.03 DAI Pay back Available to generate 6.48 DAI Generate </div>															
Vault history <table border="1" style="width: 100%; border-collapse: collapse; text-align: left;"> <thead> <tr> <th style="width: 30%;">ACTIVITY</th> <th style="width: 30%;">DATE</th> <th style="width: 40%;">TX HASH</th> </tr> </thead> <tbody> <tr> <td>Generated 6.00 new Dai from Vault</td> <td>Feb 12, 2020, 11:14:32 AM</td> <td>0x105...004187 ↗</td> </tr> <tr> <td>Generated 20.00 new Dai from Vault</td> <td>Feb 6, 2020, 7:02:22 PM</td> <td>0xd025...4074631 ↗</td> </tr> <tr> <td>Deposited 0.2000 ETH into Vault</td> <td>Feb 6, 2020, 7:02:22 PM</td> <td>0xd025...4074631 ↗</td> </tr> <tr> <td>Opened a new Vault with id #4807</td> <td>Feb 6, 2020, 7:02:22 PM</td> <td>0xd025...4074631 ↗</td> </tr> </tbody> </table>		ACTIVITY	DATE	TX HASH	Generated 6.00 new Dai from Vault	Feb 12, 2020, 11:14:32 AM	0x105...004187 ↗	Generated 20.00 new Dai from Vault	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗	Deposited 0.2000 ETH into Vault	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗	Opened a new Vault with id #4807	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗
ACTIVITY	DATE	TX HASH														
Generated 6.00 new Dai from Vault	Feb 12, 2020, 11:14:32 AM	0x105...004187 ↗														
Generated 20.00 new Dai from Vault	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗														
Deposited 0.2000 ETH into Vault	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗														
Opened a new Vault with id #4807	Feb 6, 2020, 7:02:22 PM	0xd025...4074631 ↗														

Step 4

- Congratulations! Your ETH vault is now created!

In addition to minting DAI, you can also save on Maker's platform to earn interest on your assets. We've prepared a step-by-step guide on how to save your DAI below:

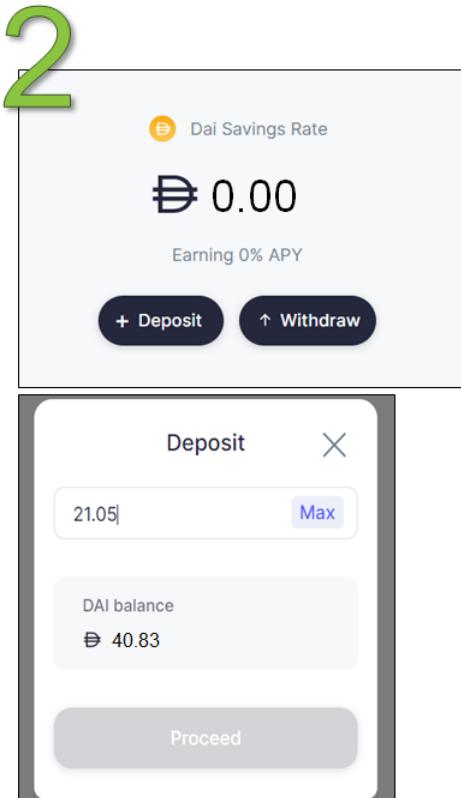
Saving your DAI

The screenshot shows the Oasis app's user interface. On the left is a dark sidebar with a vertical navigation menu. The menu items are: Save (highlighted with a green box), Borrow, Overview (highlighted with a red arrow), ETH-A 232%, ETH-A 187%, +, and Trade. Below the sidebar is the main content area. At the top of the content area is a header with two boxes: 'TOTAL COLLATERAL LOCKED \$97.52 USD' and 'TOTAL DAI DEBT 47.03 DAI'. Below this is a section titled 'Your Vaults' which contains a table of vaults.

TOKEN	Vault ID	CURRENT RATIO	DEPOSITED	AVAIL. TO WITHDRAW	DAI	Manage Vault
ETH	5179	232.20%	0.20 ETH	0.07 ETH	21.00 DAI	<button>Manage Vault</button>
ETH	4807	187.36%	0.20 ETH	0.04 ETH	26.03 DAI	<button>Manage Vault</button>

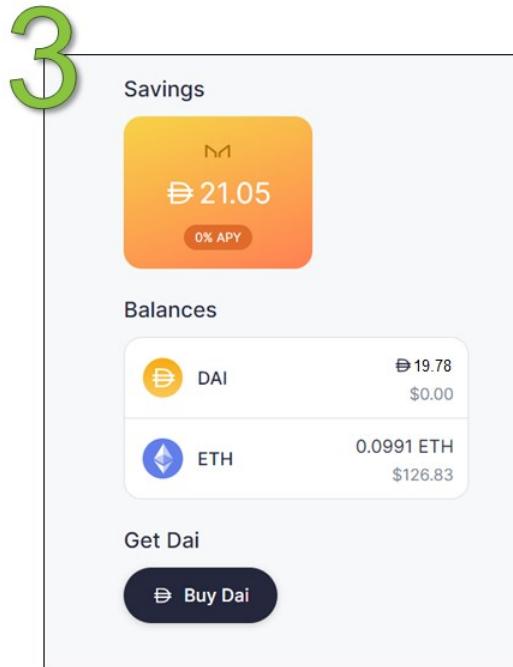
Step 1

- Navigate to the Save page on the left sidebar (<https://oasis.app/save>)



Step 2

- Click “Deposit”
- Enter the amount of DAI you wish to save
- Click “Proceed”
- Confirm in wallet



Step 3

- And it's done!
- Note: You only have one DSR account. If you were to deposit more DAI after your first deposit, it will be added to it.
- Kindly note at the time of writing (April 2021), the DSR has 0% APY.

Recommended Readings

1. Maker Protocol 101 (Maker) <https://docs.makerdao.com/maker-protocol-101>
2. Maker for Dummies: A Plain English Explanation of the Dai Stablecoin (Gregory DiPrisco)
<https://medium.com/cryptolinks/maker-for-dummies-a-plain-english-explanation-of-the-dai-stablecoin-e4481d79b90>
3. What's MakerDAO and what's going on with it? Explained with pictures. (Kerman Kohli) <https://hackernoon.com/whats-makeraodao-and-what-s-going-on-with-it-explained-with-pictures-f7ebf774e9c2>
4. How to get a DAI saving account (Ryan Sean Adams)
<https://bankless.substack.com/p/how-to-get-a-dai-saving-account>
5. Maker's Black Swan Event
<https://tokentuesdays.substack.com/p/makers-black-swan>

CHAPTER 6: DECENTRALIZED LENDING AND BORROWING

One of the biggest services offered by the financial industry is the lending and borrowing of funds, which was made possible by the concept of credit and collateralization. As of 1 April 2021, the borrowing volume increased 102 times more than a year ago, reaching \$9.7 billion.

It can be argued that the invention of commercial-scale lending and borrowing was what brought about the Renaissance age as the possibility for the less wealthy to acquire startup funds led to a flurry of economic activity. Thus, the economy began to grow at an unprecedented pace.

Entrepreneurs can borrow the upfront capital needed to establish a business by collateralizing the company. Families can get a mortgage for a house that would otherwise be too costly to buy in cash by using the house as collateral. On the other hand, the wealth accumulated can be lent out as capital to borrowers. This system reduces the risk of borrowers absconding with the borrowed funds.

However, this system requires some form of trust and an intermediary. Banks have historically taken up an intermediary role while a convoluted credit system maintains trust. In this credit system, borrowers must exhibit the ability to repay the loan to be qualified to borrow, among a laundry list of other qualifications and requirements by the banks.

Decentralized Lending and Borrowing

This has led to various challenges and shortfalls of the current lending and borrowing system, such as restrictive funding criteria, geographical or legal restrictions to access banks, high barriers to loan acceptance, and the exclusivity of only the wealthy to enjoy the benefits of low-risk high-returns lending.

In the DeFi landscape, such barriers do not exist as banks are no longer necessary. With enough collateral, anyone can have access to capital to do whatever they want. Capital lending is also no longer enjoyed only by the wealthy; everyone can contribute to a decentralized liquidity pool that borrowers can take from and pay back at an algorithmically-determined interest rate. In contrast to applying for a loan from the bank where there are stringent Know-your-customer (KYC) and Anti-money laundering (AML) policies, one only needs to provide collateral to take a loan in DeFi.

We will explore how such bankless lending and borrowing mechanisms are possible with Compound Finance and Aave, the two leading DeFi lending and borrowing protocols.

Compound



Compound Finance is an Ethereum-based, open-source money market protocol where anyone can lend or borrow cryptocurrencies frictionlessly. As of 1 April 2021, there are nine different tokens available on the Compound Platform:

1. [0x \(ZRX\)](#)
2. [Basic Attention Token \(BAT\)](#)
3. [Compound \(COMP\)](#)

4. [Dai \(DAI\)](#)
5. [Ether \(ETH\)](#)
6. [USD Coin \(USDC\)](#)
7. [Tether \(USDT\)](#)
8. [Uniswap \(UNI\)](#)
9. [Wrapped Bitcoin \(WBTC\)](#)

Do note that USDT is the only token that cannot be used as collateral because its fee structure could potentially impact Compound's liquidity mechanism.

Compound operates as a liquidity pool built on the Ethereum blockchain. Suppliers supply assets to the liquidity pool to earn interest, while borrowers take a loan from the liquidity pool and pay interest on their debt. In essence, Compound bridges the gaps between lenders who wish to accrue interest from idle funds and borrowers who want to borrow funds for productive or investment use.

In Compound, interest rates are denoted in Annual Percentage Yield (APY), and the interest rates differ between assets. Compound derives the interest rates via algorithms that take into account the supply and demand of the assets.

Essentially, Compound lowers the friction for lending/borrowing by allowing suppliers/borrowers to interact directly with the protocol for interest rates without needing to negotiate loan terms (e.g., maturity, interest rate, counterparty, collaterals), thereby creating a more efficient money market.

How much interest will you receive, or pay?

The Annual Percentage Yield (APY) differs between assets as it is algorithmically set based on the supply and demand of the various assets. Generally, the higher the borrowing demand, the higher the interest rate (APY) and vice versa.

Decentralized Lending and Borrowing

All Markets					
Market		Total Supply	Supply APY	Total Borrow	Borrow APY
 Ether ETH		\$1,571.72M -0.38%	0.13% -	\$96.19M -0.45%	2.65% -
 Dai DAI		\$1,290.69M -1.19%	4.45% +0.34	\$1,059.12M -0.69%	6.44% +0.47
 Wrapped BTC WBTC		\$1,145.97M -0.29%	0.21% +0.01	\$74.84M +1.95%	4.04% +0.04
 USD Coin USDC		\$1,026.58M +1.35%	7.80% +1.45	\$873.43M +3.12%	10.01% +1.72
 Tether USDT		\$178.70M +118%	8.01% +0.80	\$152.41M +2.14%	10.26% +0.94
 Uniswap UNI		\$132.56M +0.15%	0.20% +0.01	\$8.33M +5.02%	4.29% +0.11
 Ox ZRX		\$78.30M +0.10%	2.58% -	\$23.98M +0.03%	11.74% -0.01
 Compound Governance Token COMP		\$60.05M +3.65%	2.92% -0.20	\$22.29M -	10.90% -0.34
 Basic Attention Token BAT		\$28.94M +0.46%	2.43% -0.01	\$8.67M +0.34%	11.26% -0.01

<https://compound.finance/markets>

Using the DAI stablecoin as an example, a lender would earn 4.45% APY (as of January 2021) in a year, while a borrower would be paying 6.44% APY interest after a year.

Do I need to register for an account to start using Compound?

No, you do not need to register for an account, and that's the beauty of Decentralized Finance applications! Unlike traditional financial applications where users must go through lengthy processes to get started, Compound users do not need to register for anything.

Anyone with a supported cryptocurrency wallet such as Argent and Metamask can start using Compound immediately.

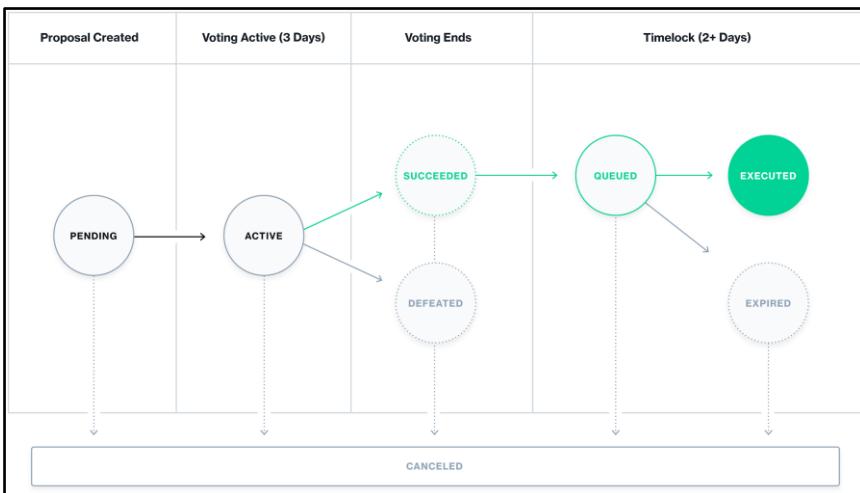
Compound Governance

Compound became progressively decentralized since its inception and has transitioned into a fully community-powered protocol via the introduction of the COMP governance token in June 2020. COMP holders can suggest,

debate, and implement changes to Compound via voting on [Compound's voting dApp](#). For more information on governance tokens, refer to [Chapter 13](#).

Examples of changes that governance can make to the Compound protocol include the addition of new assets or system parameter adjustments such as collateral factors or interest rate algorithms.

How does Compound Governance work?



There are three core components in governing the Compound protocol:

1. COMP token
2. Governance module (Governance Alpha)
3. Timelock

To table a governance proposal, an address (known as a delegate) must have more than 1% of the total COMP supply of 10,000,000 delegated to it (100,000 COMP). This stage is known as Governor Alpha.

Once submitted, there is a 3-day voting period where a minimum of 400,000 votes must be received by COMP token holders (the quorum is 4% of the total COMP supply).

Once a minimum threshold of votes has been received, passed proposals will be queued in the Timelock. There will be a 2-day grace period before passed proposals are implemented into the Compound protocol.

Users can obtain COMP tokens by buying them off the secondary markets or yield farming the COMP tokens by lending or borrowing on the Compound protocol. COMP tokens are distributed on a pro-rata basis based on the interest rates of lending and borrowing activity on Compound.

Start earning interest on Compound

To earn interest, you will have to supply assets to the protocol. As of January 2021, Compound accepts nine types of tokens.

Once you have deposited your assets into Compound, you will immediately begin to earn interest on your deposits. Interest is accrued on the amount supplied and calculated after each Ethereum block (average ~15 seconds).

Upon deposit, you will receive corresponding amounts of cTokens. If you supply DAI, you will receive cDAI, if you supply Ether, you will receive cETH, and so on. Interest is not immediately distributed but instead accrues on the cTokens, and is redeemable for the underlying asset and interest it represents.

Note: USDT is the only asset that cannot be used as collateral because of the counterparty exposure risk. As mentioned in [Chapter 2](#), users will need to trust that every USDT is fully backed 1:1 with USD and the reserve exists. Compound fears that an infinite quantity of USDT could be minted to drain assets from the protocol.²

What are cTokens?

cTokens represent your balance in the protocol and accrue interest over time. In Compound, interest earned is not distributed immediately but instead accrues in cTokens.

² (n.d.). CoinGecko. Retrieved March 23, 2021, from
<https://discord.com/channels/402910780124561410/765610989847969810/790721926372261890>

Let's go through this with an example. Assume that you have supplied 1,000 DAI on 1 January 2020, and APY has been constant at 10% throughout 2020.

On 1 January 2020, after you have deposited 1,000 DAI, you will be given 1,000 cDAI. In this case, the exchange rate between DAI and cDAI is 1:1.

On 1 January 2021, after one year, your 1,000 cDAI will now increase in value by 10%. The new exchange rate between DAI and cDAI is 1:1.1. Your 1,000 cDAI is now redeemable for 1,100 DAI.

1 Jan 2020: Deposit 1,000 DAI. Receive 1,000 cDAI. Exchange Rate:
1 cDAI = 1 DAI

1 Jan 2021: Redeem 1,000 cDAI. Receive 1,100 DAI. Exchange Rate:
1 cDAI = 1.1 DAI (cDAI value increased by 10%)

To account for the interest accrued, cTokens become convertible into an increasing amount of the underlying asset it represents over time. cTokens are also ERC-20 tokens, meaning you can easily transfer the “ownership” of supplied assets if someone wants to take over your position as a supplier.

Start borrowing on Compound

Before borrowing, you have to supply assets into the system as collateral for your loan. The more assets that you supply into Compound, the greater the amount that you can borrow. Additionally, each asset supplied has a different collateral factor that determines the amount you can borrow.

Borrowed assets are sent directly to your Ethereum wallet, and from there, you can use them as you would any cryptoasset—anything your financial strategy desires!

Do note that a portion of the interest paid by the borrower will go to its reserve - which acts as the insurance and is controlled by the Compound token (COMP) holders. Each supported asset has its reserve factor that will determine how much goes into the reserve.

Price movement of collateral asset

If you are thinking about depositing your assets as collateral to take a loan, you may wonder what happens if the value of the collateral changes? Let's see:

1. Collateral value moves up

If the value of the asset you used as collateral goes up, your collateral ratio also goes up, which is fine—nothing will happen, and you can draw a bigger loan if you would like to.

2. Collateral value moves down

On the other hand, if the collateral goes down such that your collateral ratio is now below the required collateral ratio, your collateral will be partially sold off along with an 8% liquidation fee. The process of selling off your collateral to achieve the minimum collateral ratio is known as liquidation.

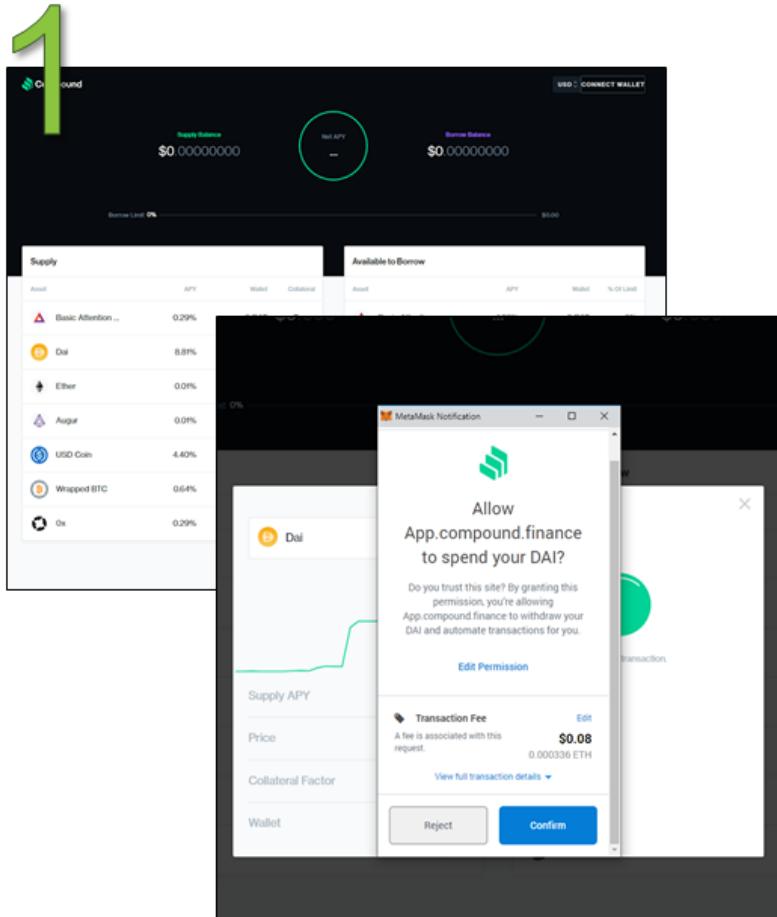
Liquidation

Liquidation occurs when the value of the collateral provided is less than the borrowed funds. Liquidation happens to ensure that there will always be excess liquidity for withdrawal and borrowing of funds, protecting lenders against default risk. The current liquidation fee is 8%.

And that's it for Compound—if you're keen to get started or test it out, we have included step-by-step guides on how to (i) supply funds to the pool and (ii) borrow from the pool. Otherwise, head on to the next section to read more on the next DeFi Dapp!

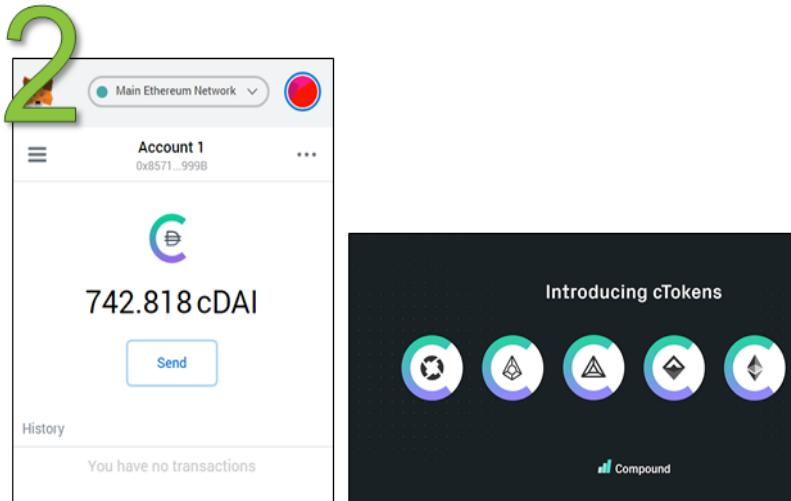
Compound.Finance: Step-by-Step Guides

Supplying funds to the pool:



Step 1

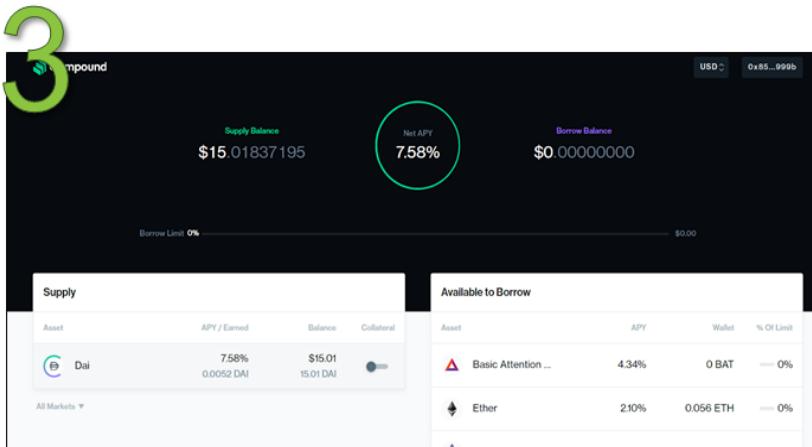
- Head over to <https://app.compound.finance>
- Connect your wallet. Follow your wallet's instructions
- Deposit cryptocurrencies into the liquidity pool (any of the 9 tokens)



Step 2

- Receive cTokens
- When you sign up for a fixed deposit, the bank will issue a fixed deposit certificate as proof of placement. Similarly, after supplying assets, you will get cTokens which represent the type and amount of assets you have deposited
- The cTokens act as a claim of deposit and record the interest you earn. Likewise, you have to use it to redeem or withdraw your assets

How to DeFi: Beginner



Step 3

- Earn Interest
- You start to earn interest the moment you deposit assets and receive cTokens in return. By holding the cTokens, the interest will accrue on your account.

Step 4

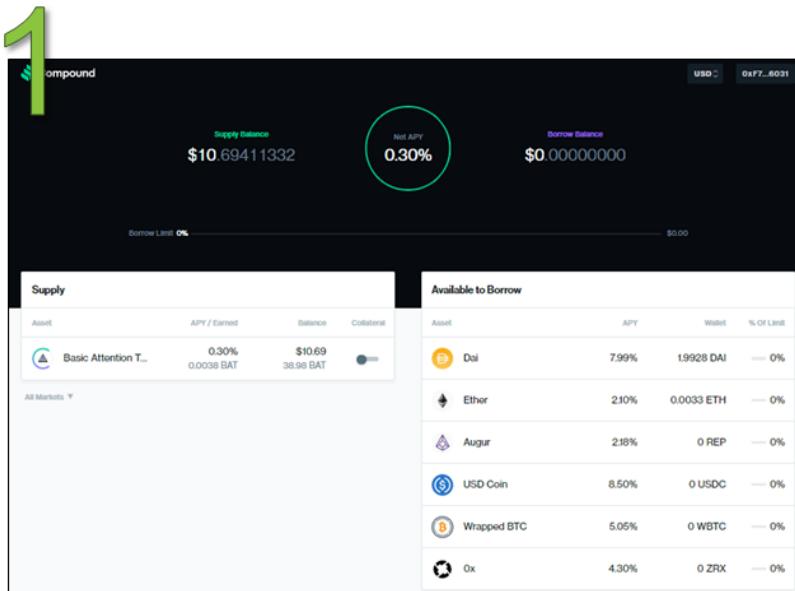
- Redeem cToken
- Over time, the interest accumulates and each cTokens is convertible into a greater value of underlying assets. You can redeem the cTokens anytime and receive the assets back to your wallet instantly.

Decentralized Lending and Borrowing

Borrowing funds from the pool:

Note:

- Before you could start borrowing, you are required to supply some assets into the compound as a form of collateral. USDT cannot be used as collateral for borrowing funds.
- Each token has its own collateral factor. A collateral factor is the ratio of how much you have to supply in order to borrow.
- You cannot supply and borrow the same token at the same time

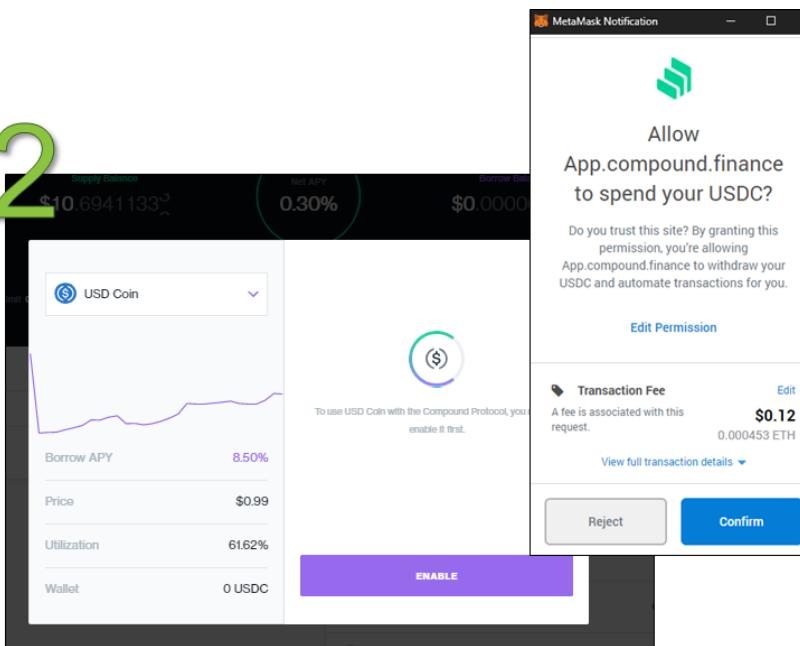


Step 1

- Go to the Compound's main page <https://app.compound.finance/>
- Choose which tokens you wish to borrow on the right bar. We chose USDC

How to DeFi: Beginner

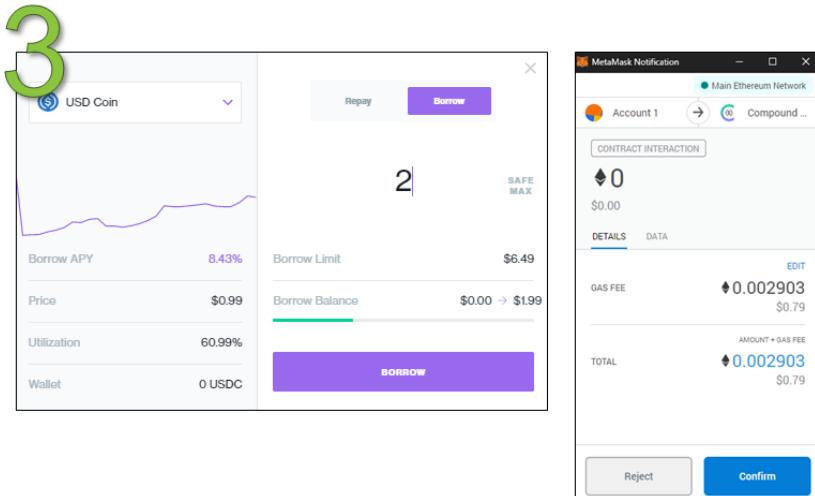
2



Step 2

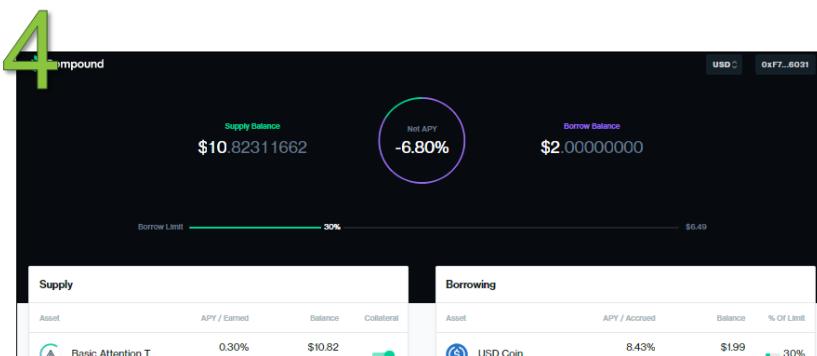
- A pop-up on USDC will appear
- Each token has to be enabled individually. You only need to do this once per token

Decentralized Lending and Borrowing



Step 3

- We entered the amount we wish to borrow. In this example, we borrowed 2 USDC
- Confirm the transaction with your wallet



Step 4

- Finished!
- You can see how much you have supplied and how much you have borrowed on Compound's main page.

Recommended Readings

1. The DeFi Series – An overview of the ecosystem and major protocols (Alethio) <https://medium.com/alethio/the-defi-series-an-overview-of-the-ecosystem-and-major-protocols-da27d7b11191>
2. Compound FAQ (Robert Leshner) <https://medium.com/compound-finance/faq-1a2636713b69>
3. DeFi Series #1 - Decentralized Cryptoasset Lending & Borrowing (Binance Research) <https://research.binance.com/analysis/decentralized-finance-lending-borrowing>
4. Zero to DeFi – A beginner’s guide to earning passive income via Compound Finance (Defi Pulse) <https://defipulse.com/blog/zero-to-defi-cdai/>
5. I took out a loan with cryptocurrency and didn’t sign a thing (Stan Schroeder) <https://mashable.com/article/defi-guide-ethereum-decentralized-finance.amp>
6. Earn passive income with Compound. (DefiZap) <https://defitutorials.substack.com/p/earn-passive-income-with-compound>

Aave



Aave is another prominent decentralized money market protocol. As of April 2021, users can lend and borrow 24 different assets on Aave. Notably, Aave offers more assets for users to lend and borrow as compared to Compound.

Both Compound and Aave operate similarly where lenders can provide liquidity by depositing cryptocurrencies into the available lending pools and earn interest. Borrowers can take loans by tapping into these liquidity pools and pay interest.

Aave is more complex than Compound as it offers more flexibility and features. If you want a deeper dive into the key differences between the protocols, you may refer to our How to DeFi: Advanced book.

Here are **eight key features** on Aave:

1. Support more assets

As of April 2021, Aave offers 24 assets for lending and borrowing. Aave has historically been quick in adding more assets to its platform.

2. Stable and variable interest rates on loan

Borrowers have the flexibility to choose between stable or variable interest rates.

3. Rate Switching

Borrowers are also able to switch between stable and variable interest rates.

4. Collateral Swap

Borrowers can swap their collateral for another asset. This helps to prevent loans from going below the minimum collateral ratio and face liquidation.

5. Repayment with collateral

Borrowers can close their loan positions by paying directly with their collateral in one transaction.

6. Flash loans

Borrowers can take up loans with zero collateral if the borrower repays the loan and any additional interest and flash loan fees within the same transaction. Flash loans are useful for arbitrage traders as they are capital-efficient in making arbitrage trades across the various DeFi Dapps.

7. Flash Liquidations

Liquidators can utilize flash loans to borrow capital as part of their liquidation bond and get that liquidation bonus without using their capital.

8. Native Credit Delegation

Borrowers may extend their credit line to other users who wish to borrow without collateral for a higher interest rate.

How much interest will you receive or pay?

Like Compound, the interest rates for both borrowers and lenders are determined algorithmically, subjected to the supply and demand for each asset.

Essentially, the higher the borrowing demand, the higher the interest rate due to less available liquidity in the pool. When this occurs, lenders will earn more.

Assets	Market size	Total borrowed	Deposit APY	Variable Borrow APR	Stable Borrow APR
USDT Coin (usdt)	\$ 35.6M	\$ 33.28M	12.72 %	24.95 %	29.95 %
Gemini Dollar (gUSD)	\$ 2.93M	\$ 2.38M	8.45 %	10.39 %	—
DAI	\$ 48.42M	\$ 39.8M	8.01 %	12.24 %	17.24 %
TrueUSD (tUSD)	\$ 5.27M	\$ 4.27M	6.47 %	7.86 %	12.86 %
USD Coin (usdc)	\$ 71.19M	\$ 60.78M	5.82 %	3.79 %	8.90 %
Binance USD (busd)	\$ 7.02M	\$ 5.65M	4.52 %	6.24 %	—
SNX	\$ 6.42M	\$ 2.3M	1.95 %	8.37 %	—
Ethereum (ETH)	\$ 179.45M	\$ 49.84M	0.87 %	3.42 %	7.27 %
sUSD	\$ 4.47M	\$ 2.04M	0.83 %	2.28 %	—
Curve DAO Token (crv)	\$ 18.25M	\$ 3.75M	0.66 %	3.19 %	—
REN	\$ 3.38M	\$ 772.46K	0.24 %	3.55 %	5.08 %

As of the time of writing (April 2021), using USDT as an example, lenders who deposit their USDT would earn 5.99% APY. Borrowers meanwhile have the option to choose a loan with variable or stable interest rate. The

variable interest rate for USDT was 3.97% APR while the stable interest rate for USDT was 11.99% APR.

Which interest rate should I choose?

Stable Annual Percentage Rate (APR)

Stable rate is similar to the fixed rate, but it can change over time if market conditions get dire. Borrowers who choose a stable rate prefer to know the exact amount of interest to be repaid and are less likely to be impacted by liquidity fluctuation in the respective liquidity pools.

Variable Annual Percentage Rate (APR)

The variable rate is algorithmically determined based on the supply and demand of an asset in the Aave protocol. The variable interest rates fluctuate to correspond to the amount of liquidity available in reserve.

Do I need to register for an account to start using Aave?

You don't need to! Aave is a decentralized lending protocol where you can connect your wallet and start lending or borrowing available assets.

Start earning interest on Aave

The mechanics are similar to Compound. You will first have to supply assets to the Aave protocol.

Upon depositing your asset, you will receive a proportional amount of aTokens that represents your underlying assets. If you supply USDT, you will receive aUSDT. If you supply YFI, you will receive aYFI and so on.

The interest will immediately accrue on your aTokens on every Ethereum block (~15 seconds). However, you will not receive your interest immediately, and you will need to redeem your aTokens to receive back your capital plus the interest accrued.

Start borrowing on Aave

Before you can start borrowing, you must deposit an asset as collateral to borrow, and the amount deposited must be higher than the amount borrowed. Unlike Compound, Aave determines how much a user can

borrow via a pre-set Loan to Value ratio (LTV), and each asset has a different LTV ranging from [15% to 80%](#).

If the LTV hits the asset's liquidation threshold, liquidators can liquidate up to 50% of your position, with an additional liquidation penalty going up to 15%, depending on the asset.

Do note that not all assets can be used as collateral. Like Compound, tokens that have a single counterparty risk exposure cannot be used as collateral as they can potentially be used to drain the protocol of all liquidity. Four assets cannot be used as collateral at the time of writing (April 2021), namely USDT, GUSD, BUSD, and sUSD.

Aave Governance and how it works?

Aave relaunched the second iteration of its protocol (Aave V2) in December 2020. Its native governance token is known as AAVE. Anyone is free to propose an idea for Aave's improvements and adjustments. Here is an overview of how one can table a proposal.

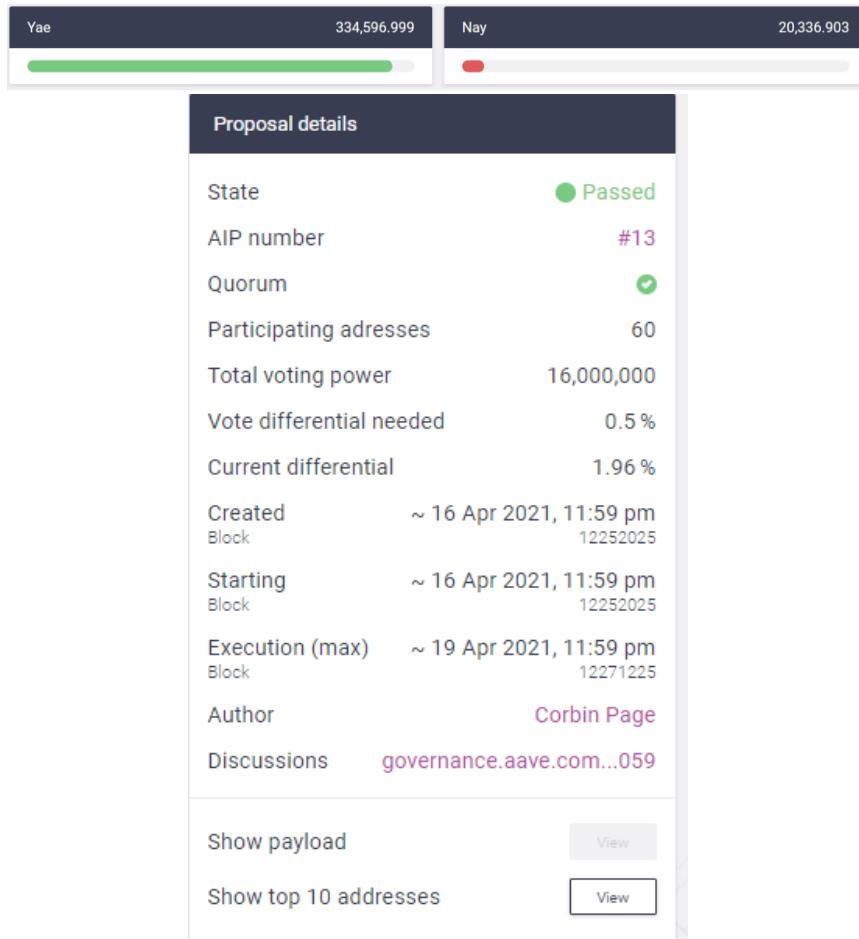
1. Prepare an Aave Request for Comments (ARC) proposal on the Aave [governance forum](#). The community will give feedback on it.
2. If the ARC is non-contentious, the submission of an Aave Improvement Proposal (AIP) can be made.
3. The AIP is submitted to the protocol for voting.

Currently, there are two types of proposal. The initial parameters varies on each type:

- Short time lock executor
 - The initial quorum is 2% (“Yes” votes $\geq 2\%$)
The percentage of “Yes” votes out of total voting power (that is derived from the maximum supply of 16,000,000)
 - The vote differential is 0.5%. (“Yes” votes - “No” votes $\geq 0.5\%$)
The minimum difference between “Yes” and “No” votes out of total voting power.
- Long time lock executor
 - The initial quorum is 20% (“Yes” votes $\geq 20\%$)
 - The vote differential is 15%. (“Yes” votes - “No” votes $\geq 15\%$)

Decentralized Lending and Borrowing

Let's take a look at [a proposal voted on Aave](#):



Here we can see that:

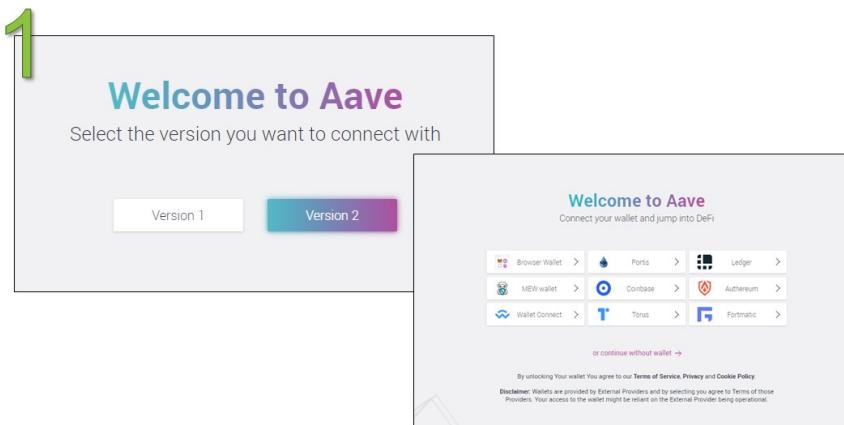
- The proposal is a short time lock executor type.
- Thus the vote differential needed is 0.5% while the quorum needed is 2%
- “Yes” votes amounted to 334,597, or 2.1% of the voting power. This passed the quorum criteria.
- “No” votes amounted to 20,337, or 0.127%
- This implies the current vote differential is 1.96% (2.1% - 0.127%)
- The proposal is passed.

However, it must be noted that the quorum is dynamic and can change based on the quorum + differential of votes for/against an AIP. Using the same proposal above as an example:

- Assuming the “Yes” votes remain the same at 2.1%
- But the “No” votes increased to 1.8%
- Then the current vote differential would become 0.3%, which is lower than the differential needed of 0.5%.
- In order for the vote to pass, the yes vote will have to increase from 2.1% to 2.3% (1.8% + 0.5%).
- Thus, this implies the quorum threshold would increase to 2.3%.

Aave: Step-by-Step Guide

Supplying funds from the pool:



Step 1

- Go to <https://app.aave.com/>
- Connect the wallet of your choice. We used Metamask in this example (Browser Wallet)

Decentralized Lending and Borrowing

The screenshot shows the Aave v2 interface. At the top, there is a navigation bar with tabs: MARKETS, DASHBOARD, DEPOSIT (which is highlighted with a red box), BORROW, SWAP, STAKE, GOVERNANCE, and MORE. To the right of the tabs is a dropdown menu labeled 'AAVE market'. Below the navigation bar is a search bar with placeholder text 'Search' and a 'Stable Coins' button. The main content area is titled 'Deposit' and displays a table of available assets:

Asset	Your wallet balance	APY %
DAI	168.94 DAI	6.54 %
USDC	31.59 USDC	5.69 %
TUSD	0.00 TUSD	6.91 %
USDT	0.00 USDT	7.64 %
SUSD	0.00 SUSD	41.39 %

Step 2

- Select “Deposit” at the top header
- You will see a list of available assets that you can deposit, along with their respective APYs

3

How much would you like to deposit?
Please enter an amount you would like to deposit. The maximum amount you can deposit is shown below.

Available to deposit 168.937073 DAI
100 MAX

Continue Go back

Deposit overview
These are your transaction details. Make sure to check if this is correct before submitting.

Amount 100 DAI \$ 100.42208
1 Approve 2 Deposit 3 Finished
Approve Please approve before depositing Submit
Go back

MetaMask Notification
Allow https://app.aave.com to spend your DAI?
Do you trust this site? By granting this permission, you're allowing https://app.aave.com to withdraw your DAI and automate transactions for you.
Edit Permission
Transaction Fee Edit
A fee is associated with this request. \$5.00 0.003767 ETH
View full transaction details Reject Confirm

MetaMask Notification
CONTRACT INTERACTION Ethereum Mainnet CG Training 0x7d27...c7...
DETAILS DATA
GAS FEE 0.015884 \$21.13
TOTAL 0.015884 \$21.13
AMOUNT + GAS FEE
Reject Confirm

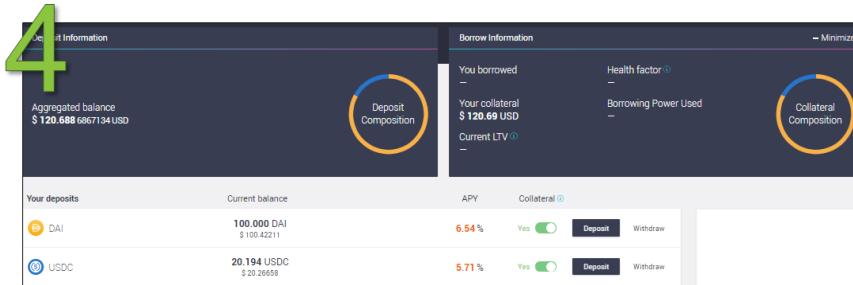
Congrats!
Your action has been successfully executed

Amount 100 DAI \$ 100.42208
1 Approve 2 Deposit 3 Finished
Success! Dashboard
Approve Confirmed Etherscan Deposit Confirmed Etherscan

Step 3

- We chose to supply 100 DAI to the protocol.
- Before you can deposit, you will need to sign approval for the protocol to spend your DAI (this will incur gas fee).
- Upon confirmation, you can then deposit your 100 DAI. You will need to sign another transaction to transfer over your deposit.
- Done! Your asset is now deposited.

Decentralized Lending and Borrowing

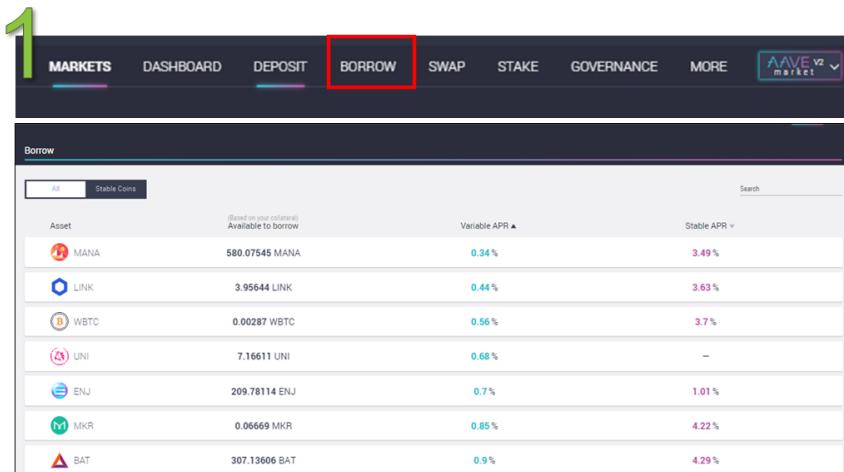


Step 4

- You can see your deposited funds on the “Dashboard” tab.

Borrowing funds from the pool:

Note: Before you can start borrowing, you are **required to supply some assets to Aave as a form of collateral**. Each token has its Loan-to-Value (LTV) ratio which is the ratio of how much you have to supply in order to borrow.



Step 1

- Select “Borrow” on the bar top.
- You’ll see a list of available assets that you can borrow with their respective APRs.

2

How much would you like to borrow?

Please enter an amount you would like to borrow. The maximum amount you can borrow is shown below.

Available to borrow	209.781364 ENJ
35	MAX

Safer Risky New health factor 6.39

Continue

[Go back](#)

Step 2

- You can choose which asset you wish to borrow. We chose to borrow 35 ENJ.
- Click “Continue”.

3

Please select your interest rate

Choose either stable or variable APR for your loan. Please click on the desired rate type and read the info box for more information on each option.

Stable APR 0.84 %	Variable APR 0.59 %
----------------------	------------------------

Borrow overview

These are your transaction details. Make sure to check if this is correct before submitting.

Amount	35 ENJ
Interest (APR)	0.84 %
Interest rate type	Stable
New health factor	6.52

1 Borrow	2 Finished
Borrow Please submit to borrow	
Submit	

Borrow overview

These are your transaction details. Make sure to check if this is correct before submitting.

Amount	35 ENJ
Interest (APR)	0.59 %
Interest rate type	Variable
New health factor	6.52

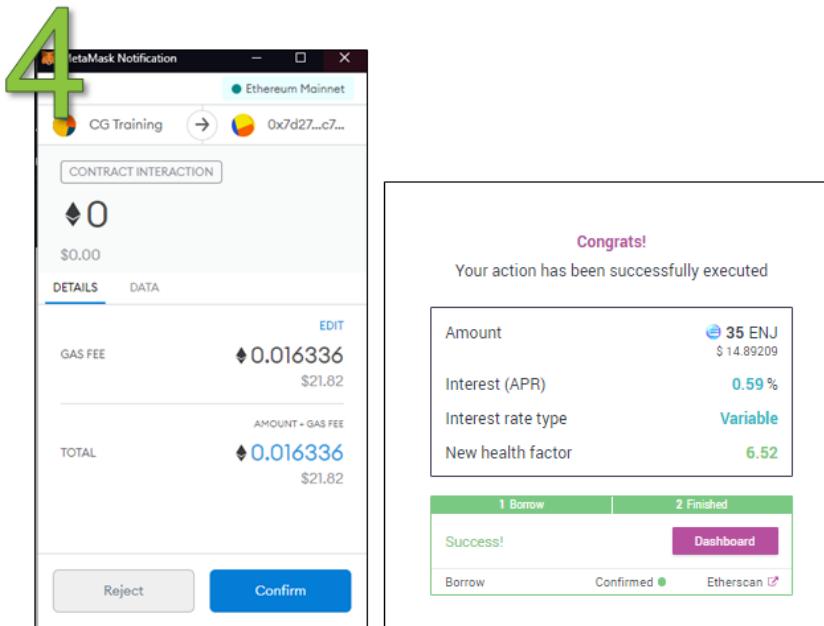
1 Borrow	2 Finished
Borrow Please submit to borrow	
Submit	

[Go back](#)

Step 3

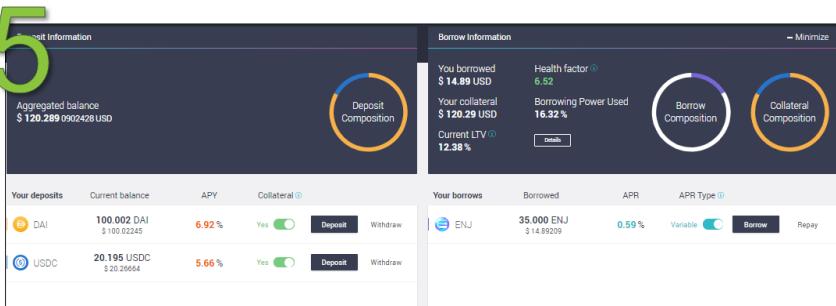
- You will have the option to choose between Stable APR or Variable APR. We chose variable APR because it is lower but that's up for you to decide.

Decentralized Lending and Borrowing



Step 4

- Approve your transaction.
- Success! You have successfully borrowed an asset (in this case, we borrowed 35 ENJ).



Step 5

- You can see your borrowed asset on the “Dashboard” tab

Recommended Readings

1. Aave FAQ <https://docs.aave.com/faq/>
2. The Aave Protocol V2 (Stani Kulechov)
<https://medium.com/aave/the-aave-protocol-v2-f06f299cee04>
3. Aave's aTokens are Latest DeFi Lego Heading to Layer 2 (The Defiant) <https://thedefiant.io/aaves-atokens-are-latest-defi-lego-heading-to-layer-2/>
4. Earning passive income from cryptocurrency in 2020: Market Review (Aave in the list) <https://hackernoon.com/earning-passive-income-from-cryptocurrency-in-2020-market-review-kzt3xom>

CHAPTER 7: DECENTRALIZED EXCHANGES (DEX)

While Centralized Exchanges (CEXs) have plenty of liquidity and allow large trades to happen, it still carries many risks because users do not hold custody of their assets in exchanges. For example, in September 2020, KuCoin suffered a \$281 million hack after a security breach.³

More people are realizing these risks and are turning to Decentralized Exchanges (DEXs). DEXs work by using smart contracts and on-chain transactions to reduce or eliminate the need for an intermediary.

Types of DEX

There are two types of DEXs:

- Order book-based DEXs
- Liquidity pool-based DEXs.

Some popular Decentralized Exchanges include Uniswap, Kyber Network, Curve Finance, dYdX, and SushiSwap. Order book-based DEXs like dYdX and Deversifi operate similarly to CEXs where users can set buy and sell orders at either their chosen limit prices or at market prices. The main difference between the two is that for CEXs, assets for the trade would be

³ (2020, September 25). Over \$280M Drained in KuCoin Crypto Exchange Hack - CoinDesk. Retrieved March 13, 2021, from <https://www.coindesk.com/hackers-drain-kucoin-crypto-exchanges-funds>

held on the exchanges' wallets, whereas for DEXs, assets for the trade would be held on users' wallets.

On the other hand, liquidity pool DEXs such as Uniswap and Balancer lets users become the market makers by providing liquidity to a pair or pool of assets. Users deposit their assets and become liquidity providers, earning a small fee for each swap transaction performed for that particular pool. However, the mechanisms behind these types of DEXs, which mainly use an Automated Market Maker mechanism, have their own flaws where users may suffer Impermanent Loss as a liquidity provider.

Limitations of DEX

DEXs have certain limitations as well. Here are some of the limitations:

1. Lower liquidity

The majority of crypto trade still takes place in centralized exchanges. Historically, order books on CEXs have been deeper when compared to DEXs, and traders can thus get better prices when making trades on CEXs.

With lower liquidity, trades on DEX may suffer higher slippage and worse price execution when compared to CEX. However, with the growing popularity of DEXs, liquidity on many popular trading pairs has increased significantly on DEXs, and trade executions on DEXs are at times as competitive as CEXs.

2. Limited features

Centralized exchanges include many advanced trading features such as limit orders, stop-loss orders, trailing stops, and so on. Most of these trading features are not available on DEXs.

Some DEXs now offer limit orders, allowing for a better trading experience. However, a growing number of DEXs are looking to introduce these advanced trading features to compete more effectively against CEXs.

3. Blockchain Interoperability

Most DEXs today only allow traders to swap tokens within the same blockchain ecosystem. Ethereum-based DEXs, for example, only allow users to trade Ethereum and ERC-20 tokens. It does not permit traders to trade tokens issued on other blockchains like Polkadot or Cosmos. CEXs allow users to trade tokens on various blockchains easily. There are efforts to build cross-chain DEXs, and in the future, trading tokens across multiple blockchains on a DEX will be possible.

4. Costs

DeFi has now become popular, and this has resulted in a congested Ethereum network. The congestion on Ethereum has allowed gas costs to increase significantly. Making a trade on a DEX can be a costly transaction, especially during peak periods.

Limitations aside, there's a growing demand for DEXs, and DEXs are still in their infancy stage. We will be dissecting DEXs into digestible bits, such as the mechanisms behind how it works, the type of transactions you can perform, and step-by-step guides on how to get started.

Uniswap



Uniswap is a decentralized token exchange protocol built on Ethereum that allows direct swapping of tokens without the need to use a centralized exchange. When using a centralized exchange, you will need to deposit tokens to an exchange, place an order on the order book, and then withdraw the swapped tokens.

On Uniswap, you can simply swap your tokens directly from your wallet without having to go through the three steps above. All you need to do is send your tokens from your wallet to Uniswap's smart contract address, and

you will receive your desired token in return in your wallet. There is no order book and the token exchange rate is determined algorithmically. All this is achieved via liquidity pools and the automated market maker mechanism.

Liquidity Pools

Liquidity pools are token reserves that sit on Uniswap's smart contracts and are available for users to exchange tokens with. For example, using the ETH-DAI trading pair with 100 ETH and 20,000 DAI in the liquidity reserves, a user that wants to buy ETH using DAI may send 202.02 DAI to the Uniswap smart contract to get 1 ETH in return. Once the swap has taken place, the liquidity pool is left with 99 ETH and 20,202.02 DAI.

Liquidity pools' reserves are provided by liquidity providers who are incentivized to obtain a proportionate fee of Uniswap's 0.3% transaction fee. This fee is charged for every token swap on Uniswap.

Anyone can be a liquidity provider - the only requirement is that one needs to provide ETH and the quoted trading token. As of January 2021, over [2.3 million ETH have been locked](#) into Uniswap. The amount of reserves held by a pool plays a huge role in determining how prices are set by the Automated Market Maker Mechanism.

Automated Market Maker Mechanism

Prices of assets in the pool are algorithmically determined using the Automated Market Maker (AMM) algorithm. AMM works by maintaining a Constant Product based on the amount of liquidity on both sides of the pool.

Let's continue the ETH-DAI liquidity pool example, using 100 ETH and 20,000 DAI. To calculate the Constant Product, Uniswap will multiply both these amounts together.

$$\begin{array}{lcl} \text{ETH liquidity (x)} & * & \text{DAI liquidity (y)} \\ 100 & * & 20,000 \end{array} = \text{Constant Product (k)}$$
$$= 2,000,000$$

Using AMM, at any given time, the Constant Product (k) must always remain at 2,000,000. If someone buys ETH using DAI, ETH will be removed from the liquidity pool while DAI will be added into the liquidity pool.

Decentralized Exchanges (DEX)

The price for this ETH will be determined asymptotically. Therefore, the larger the order, the larger the premium that is charged. Premium refers to the additional amount of DAI required to purchase 1 ETH compared to the original price of 200 DAI per ETH.

The table on the next page further elaborates the asymptotic pricing and the movement of liquidity when orders to purchase ETH are made.

As can be seen from the table, the larger the amount of ETH that a user wishes to buy, the larger the premium that will be charged. This ensures that the liquidity pool will never run dry.

How to get a token added on Uniswap?

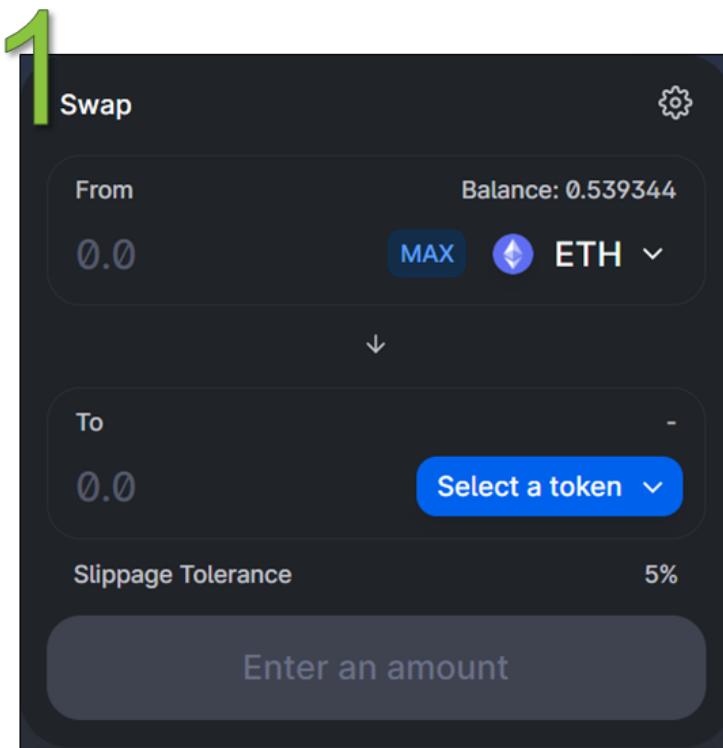
Unlike centralized exchanges, Uniswap as a decentralized exchange does not have a team or gatekeepers to evaluate and decide on which tokens to list. Instead, any ERC-20 token can be listed on Uniswap by anyone and be traded as long as liquidity exists for the given pair. All a user needs to do is interact with the platform to register the new token and a new market will be initialized for this token.

And that's it for Uniswap - if you're keen to get started or test it out, we've included step-by-step guides on how to (i) swap tokens, (ii) provide liquidity, and (iii) stop providing liquidity. Otherwise, head on to the next section to read more on the next DeFi dapp!

<i>ETH Purchased</i>	<i>Cost per ETH in DAI</i>	<i>Total Cost in DAI</i>	<i>Premium</i>	<i>New DAI Liquidity</i>	<i>New ETH liquidity</i>	<i>Product (k)</i>
1	202.02	202.02	1.01%	20,202.02	99	2,000,000
5	210.52	1,052.63	5.26%	21,052.63	95	2,000,000
10	222.22	2,222.22	11.11%	22,222.22	90	2,000,000
50	400	20,000	200%	40,000	50	2,000,000
75	800	60,000	400%	80,000	25	2,000,000
99	20,000	1,980,000	10,000%	2,000,000	1	2,000,000
100	Infinity	Infinity	Infinity	Infinity	0	2,000,000

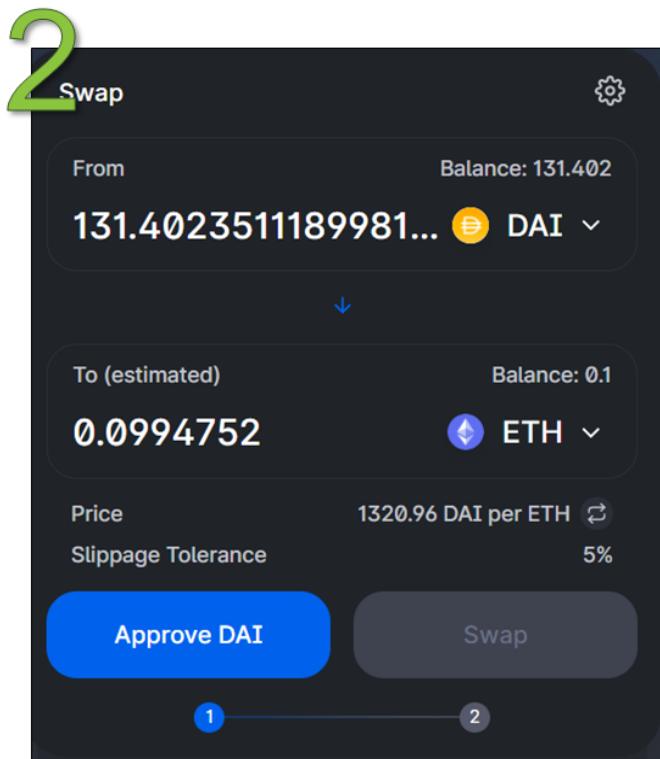
Uniswap: Step-by-Step Guide

Swapping Tokens



Step 1

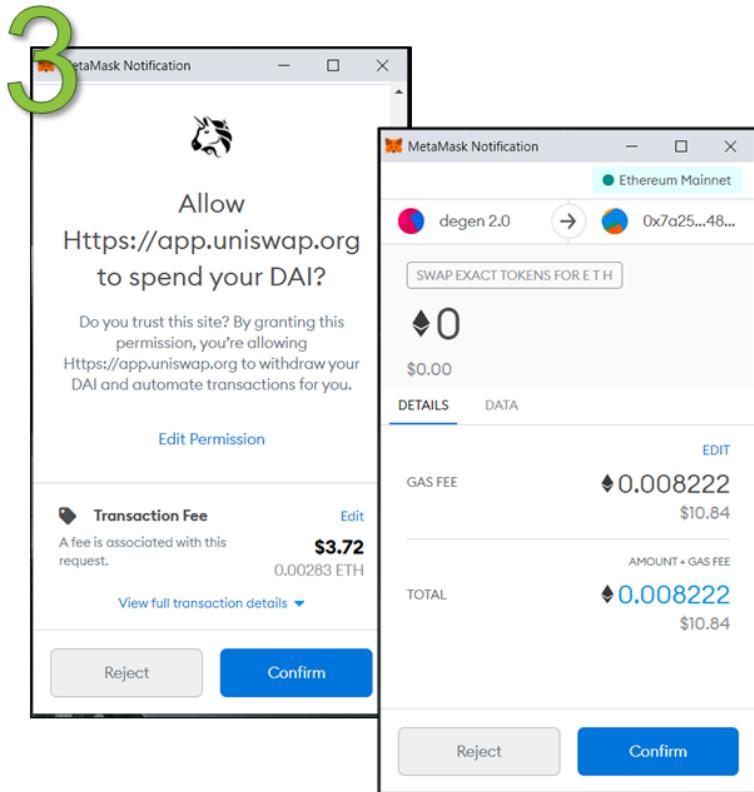
- Head on over to <https://app.uniswap.org/> and click swap token
- To start using Uniswap, you will need to connect to a wallet. You may connect your Metamask wallet. Connecting your wallet is free, all you need to do is sign a transaction



Step 2

- After connecting your wallet, choose which tokens you would like to trade. In this example, we are using DAI to buy ETH

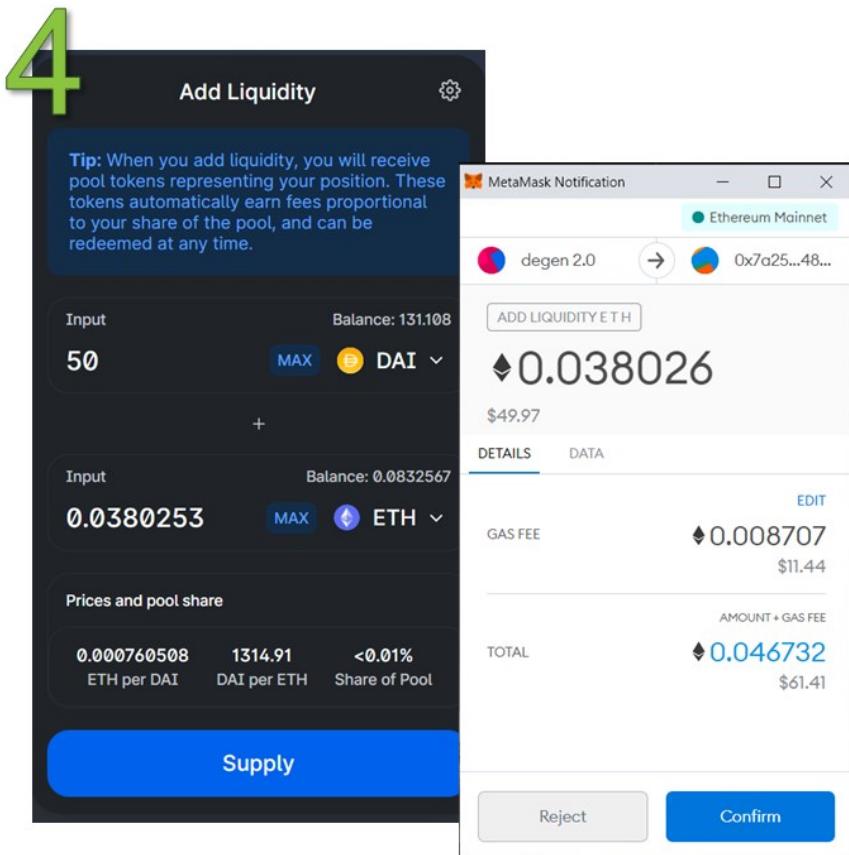
Decentralized Exchanges (DEX)



Step 3

- If it's your first time transacting this token, you'll need to unlock it by paying a small fee
- You'll be prompted with another transaction
- Once your transaction is confirmed, you'll have your ETH!

Provide Liquidity

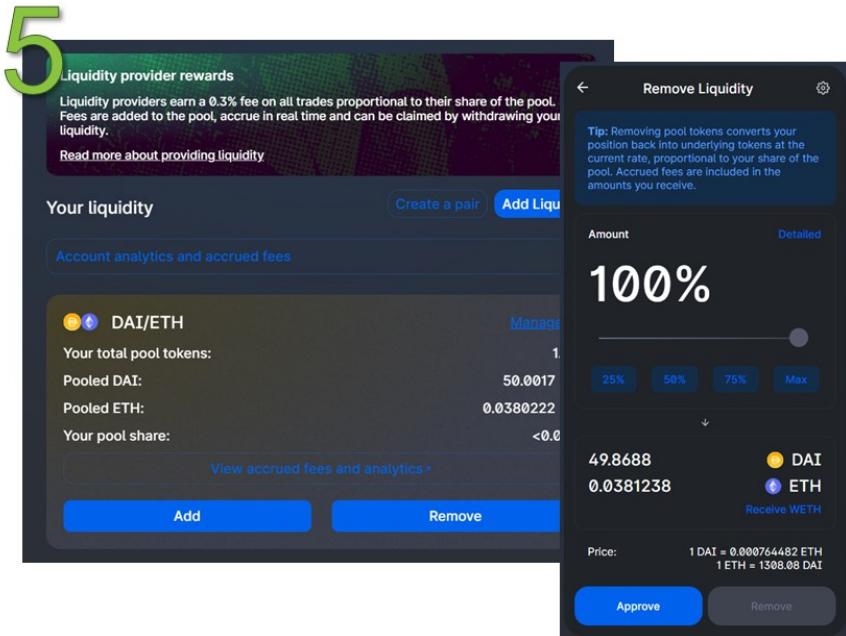


Step 4

- Go to Pool and click ‘Add Liquidity’ before filling in the amount of liquidity you wish to provide. In this case, we’re providing 50 DAI worth of liquidity + 0.0380 ETH
- Note: you must have an equivalent amount of ETH to provide liquidity for that token
- After clicking add liquidity, you’ll be shown the amount of pool tokens you will receive and a prompt to sign another transaction
- Once that’s done, you’re now confirmed as a liquidity provider and stand to earn a proportionate amount of the exchange fees

Decentralized Exchanges (DEX)

Withdraw Liquidity



Step 5

- What if you don't want to provide liquidity anymore?
- Go back to the pool to view the pairs you are providing liquidity for and select remove liquidity
- As you can see, we'll be getting back 0.13 DAI less but receiving slightly more ETH
- Note that the ratio of our ETH and DAI is now different, so that's one of the caveats with liquidity pools. If we had removed it later, we might have a very different proportion of DAI to ETH
- The pool tokens represent your share of the liquidity pool. When you remove your liquidity, you will be burning the pool tokens to get back your DAI and ETH.

Recommended Readings

1. Getting Started (Uniswap) <https://docs.uniswap.io/>
2. The Ultimate Guide to Uniswap. (DefiZap)
<https://defitutorials.substack.com/p/the-ultimate-guide-to-uniswap>
3. A Graphical Guide for Understanding Uniswap (EthHub)
<https://docs.ethhub.io/guides/graphical-guide-for-understanding-uniswap>
4. Uniswap — A Unique Exchange (Cyrus Younessi)
<https://medium.com/scalar-capital/uniswap-a-unique-exchange-f4ef44f807bf>
5. What is Uniswap? A Detailed Beginner's Guide (Bisade Asolo)
<https://www.mycryptopedia.com/what-is-uniswap-a-detailed-beginners-guide/>
6. Are Uniswap's Liquidity Pools Right for You? (Chris Blec)
<https://defiprime.com/uniswap-liquidity-pools>
7. Understanding Uniswap Returns (Pintail)
<https://medium.com/@pintail/understanding-uniswap-returns-cc593f3499ef>
8. UniSwap Traction Analysis (Ganesh)
<https://www.covalenthq.com/blog/understanding-uniswap-data-analysis/>
9. A Deep Dive into Liquidity Pools (Rebecca Mqamelo)
<https://blog.zerion.io/liquidity-pools-8ac8cf8cf230>

DEX Aggregators

There are multiple DEXs in the market today, each with its separate liquidity pools. Traders who want to make sufficiently large token swaps may incur high slippage and high price premium. The amount of slippage incurred by traders depends on the amount of liquidity available in each DEX. To minimize the slippage, traders may split the trades into smaller parts and route them onto the separate DEXs.

Splitting and routing large trades across several DEXs in a manual way is a very cumbersome process. Fortunately, traders can make use of DEX aggregators to simplify things and save gas. As its name suggests, DEX aggregators pool liquidity from the various DEXs in the market to help traders execute large trades at the best price. With DEX aggregators, traders can automatically split their large trades into smaller parts and route them to the relevant DEXs for the best execution price.

Examples of DEX aggregators include 1inch, Paraswap, and Matcha. We will be taking a closer look at 1inch, one of the most widely used DEX aggregators.

1inch



1inch is a DEX aggregator that helps users discover the best trade prices for tokens. At the time of writing (April 2021), there are over 40 sources of liquidity on 1inch. Instead of swapping tokens from a single DEX's liquidity pool, 1inch will aggregate liquidity across the various liquidity pools and suggest the most efficient token trade route.

By routing a single transaction through multiple liquidity pools instead of only one liquidity pool, traders making large trades can ensure they get the best price by minimizing price slippage. Traders also take advantage of gas savings by bundling multiple transactions across the various DEXs into a single transaction on 1inch.

To achieve the most optimum trading route for the best price execution and the lowest cost, 1inch makes use of their proprietary routing algorithm, Pathfinder. Pathfinder searches across various DEXs such as Uniswap, Balancer, and also 1inch's own liquidity protocol (formerly known as Mooniswap), before providing a recommended route for the best trade prices.

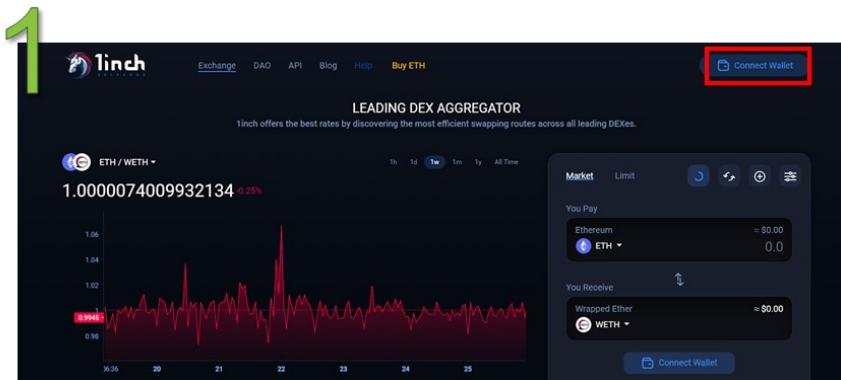
The fees charged to use 1inch depends on the fees charged by the underlying DEX. For example, Uniswap charges a 0.3% trading fee, with the fee flowing to Uniswap liquidity providers.

1inch recently launched its governance token, 1INCH, on 25 December 2020. Users holding the 1INCH governance token can participate in 1inch governance and vote on decisions affecting the future of 1inch. For example, token holders can vote on the swap fee charged on 1inch's liquidity pool, governance reward, referral reward, and more.

That's pretty much it for 1inch! If you are looking to try out 1inch, we have included a step-by-step guide on how to perform a transaction on 1inch. Otherwise, head on over to the next section to learn more about the next DeFi Dapp.

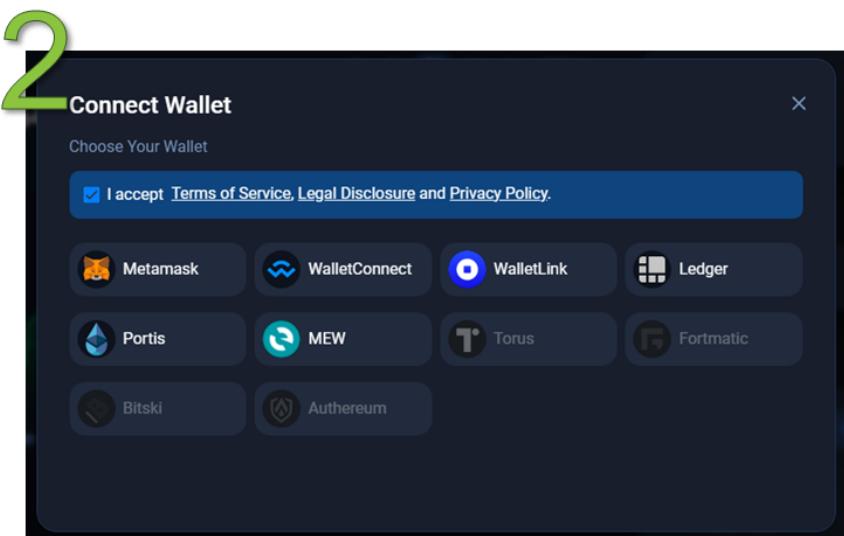
Decentralized Exchanges (DEX)

1inch: Step-by-Step Guide



Step 1

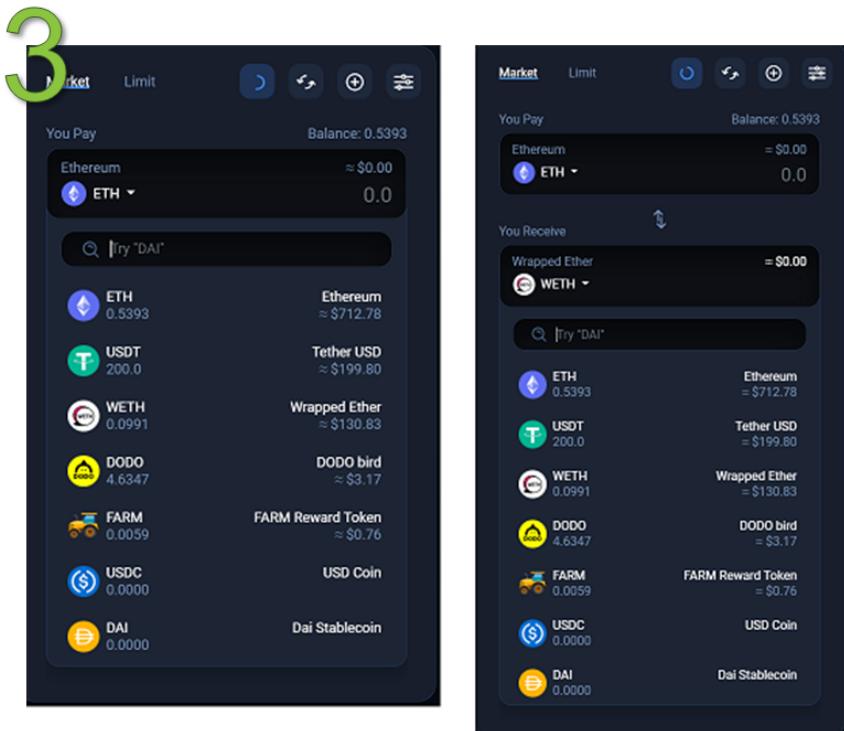
- Go to <https://1inch.exchange/>
- Click “Connect Wallet” on the top right corner



Step 2

- Click on the tick box to accept the Terms & Conditions
- Choose which wallet to connect

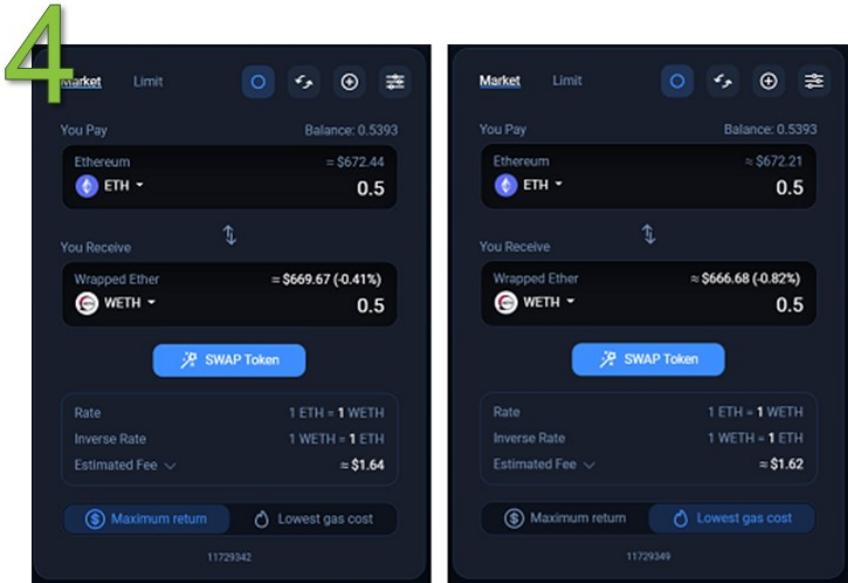
How to DeFi: Beginner



Step 3

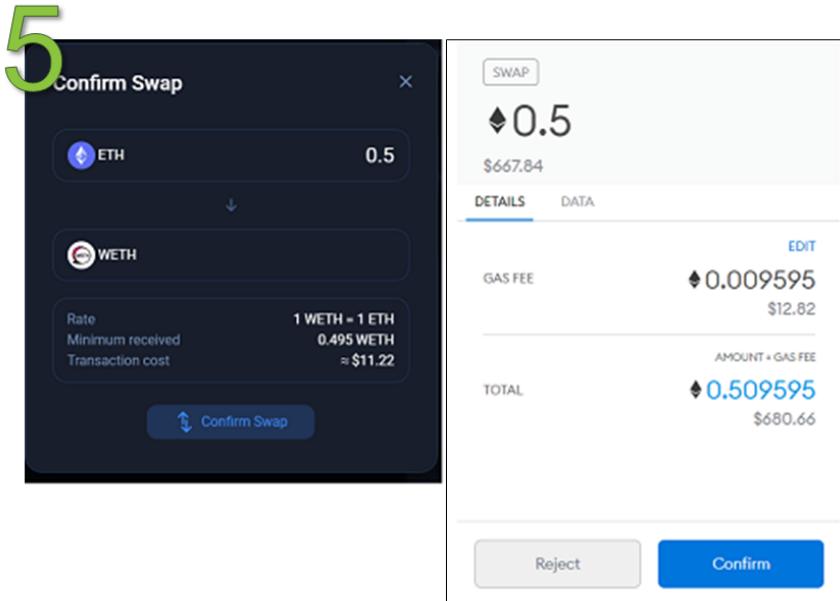
- Select which asset pair to trade
- For this example, we will choose to swap Ether for Wrapped Ether

Decentralized Exchanges (DEX)



Step 4

- Enter the amount of tokens you would like to swap
- 1inch will show you the amount of tokens you would receive in exchange
- Note that you may choose to swap tokens based on maximum return or lowest gas costs. To choose an option, simply click on the corresponding box.
- If you are satisfied with the prices, click 'SWAP Token' to submit your transaction.
- For limit orders, you have to additionally enter your desired price and expiry date to perform the transaction.



Step 5

- You will be shown the rates, minimum tokens that may be received and the transaction costs. Click ‘Confirm Swap’ to proceed with the transaction.
- Confirm the transaction through your wallet
- Once the transaction has been successfully confirmed, you’re done!

Recommended Readings

1. 1inch Exchange Review <https://defirate.com/1inch/>
2. 1inch Exchange, Mooniswap and Chi GasToken: The ultimate review and guide (Angela Wang) <https://boxmining.com/1inch-mooniswap-chi-gastoken/>
3. A beginner’s guide to trading on 1inch <https://1inch-exchange.medium.com/a-beginners-guide-to-trading-on-1inch-693d9f801a51>

CHAPTER 8: DECENTRALIZED DERIVATIVES

A derivative is a contract whose value is derived from another underlying asset such as stocks, commodities, currencies, indexes, bonds, or interest rates. Each type of derivative, whether futures, options or swaps serves a different purpose, and each investor buys or sells them for various reasons.

Some of the reasons investors trade derivatives are to hedge themselves against the volatility of the underlying asset, speculate on the directional movement of the underlying asset, or leverage their holdings. Derivatives are extremely risky, and one must be equipped with strong financial knowledge and strategies when trading them.

The market capitalization of DeFi derivatives Dapps is \$5.82 Billion, or 8.2% of the DeFi ecosystem. Decentralized derivatives have struggled to take off due to the high gas fees on Ethereum. As such, the market capitalization figure is relatively low compared to other DeFi sectors, such as the lending market (\$10.68 billion).

Although there are many DeFi derivatives protocols, we will be looking at the major ones, which are Synthetix and Opyn.

Synthetix



SYNTHETIX

Synthetix is exactly as the name sounds, a protocol for Synthetic Assets (called Synths) on Ethereum. There are two parts to Synthetix—Synthetic Assets (**Synths**) and its exchange, **Synthetix.Exchange**. Synthetix allows for the issuance and trading of Synths.

What are Synthetic Assets (Synths)?

Synths are assets or a mixture of assets that have the same value or effect as another asset. Synths track the value of underlying assets and allow exposure to the assets without the need to hold the actual asset.

There are currently two different types of Synths - Normal Synths and Inverse Synths. Normal Synths are positively correlated with the underlying assets, while Inverse Synths are negatively correlated with the underlying assets.

An example of a Synthetic Asset is Synthetic Gold (sXAU) which tracks the price performance of gold. Synthetix tracks real-world asset prices by utilizing the services of Chainlink, a smart contract oracle that obtains price feed from several trusted third-party sources to prevent tampering.

An example of an Inverse Synthetic Asset is Inverse Bitcoin (iBTC) which tracks the inverse price performance of Bitcoin. There are three key values related to each Inverse Synths - the entry price, lower limit, and upper limit.

Let's consider Inverse Synthetic Bitcoin (iBTC) as an example. Assume that at the time of creation, Bitcoin (BTC) is priced at \$10,600 - this will be the entry price. If Bitcoin moves down \$400 to \$10,200, the iBTC Synth will be worth an additional \$400 and priced at \$11,000. The opposite will also be

true. Conversely, if Bitcoin moves up to \$11,000, the iBTC Synth will now be worth \$10,200.

Inverse Synths trade in a range with a 50% upper and lower limit from the entry price. This places a cap on the maximum profit or loss you can obtain on Inverse Synths. Once either of the limits is reached, the tokens' exchange rates are frozen and the positions liquidated. Once disabled and liquidated, these Inverse Synths can only be exchanged at Synthetix.Exchange at those fixed values. They are then reset with different limits.

Why Synthetic Assets?

As mentioned above, Synths give traders price exposure to the asset without the need to hold the underlying asset. Compared to traditional gold brokerages, Synthetic Gold (sXAU) allows traders to participate in the market with much less hassle (no sign-ups, no traveling, no middleman, etc.).

Synths have another utility—they can be traded frictionlessly between one another, meaning Synthetic Gold can be switched for Synthetic JPY, Synthetic Silver, or Synthetic Bitcoin easily on [Synthetix.Exchange](#). This also means that anyone with an Ethereum wallet now has open access to any real-world asset!

How are Synths Created?

The idea behind the creation of Synths is similar to the creation of DAI on Maker. You first have to stake ETH as collateral on Maker's smart contract before being allowed to create DAI based on the collateral posted.

For Synths, you first need to stake the Synthetix Network Token (SNX), which acts as the collateral backing the entire system. SNX is less liquid compared to ETH, and its price is generally more volatile. To counter that, a large minimum initial collateral of 500% is needed on Synthetic compared to the minimum 150% initial collateral needed on Maker.

This means that to mint \$100 worth of Synthetic USD (sUSD), you will need a minimum of \$500 worth of SNX as collateral.

Note: As of January 2021, the only Synth that users can mint is sUSD.

Minting of Synths is a fairly intricate system. It entails the staker taking on debt, the levels of which dynamically fluctuate according to the total value of Synths in the global debt pool. For example, if 100% of the Synths in the system were synthetic Ethereum (sETH), and the price of ETH doubles, everyone's debt would double, including the staker's debt as well.

Once minted, these Synths tokens can be traded on Synthetix.Exchange or on Decentralized Exchanges like Uniswap.

If you want to trade Synths but don't want to take on Debt or mint your own Synths, you can also buy them from a decentralized exchange.

What Assets do Synths Support?

At the point of writing (April 2021), Synths support the following five major asset classes ([full list](#)):

- (i) **Cryptocurrencies:** Ethereum (ETH), Bitcoin (BTC), Dash (DASH), Cardano (ADA), EOS (EOS), Ethereum Classic (ETC), Monero (XMR), Binance Coin (BNB), Tezos (XTZ), Tron (TRX), Litecoin (LTC), Chainlink (LINK), Ripple (XRP), Ren (REN), Aave (AAVE), Compound (COMP), Uniswap (UNI), Yearn Finance (YFI) and Polkadot (DOT)
- (ii) **Commodities:** Gold (XAU), Silver (XAG) and Oil (OIL)
- (iii) **Fiat Currencies:** USD, AUD, CHF, JPY, EUR, and GBP
- (iv) **Indexes:** Centralised Exchange Index (CEX), FTSE 100 Index (FTSE), Nikkei 225 Index (NIKKEI) and DeFi Index (DEFI)
- (v) **Stocks:** Tesla (TSLA)

Index Synths

One of the interesting Synths available on Synthetix is the Index Synths. At the time of writing (April 2021), there are four different Index Synths, namely sCEX, sDEFI, sFTSE, and sNIKKEI.

Index Synths provide traders with exposure to a basket of tokens without the need to purchase all the underlying tokens. The index will mirror the overall performance of the underlying tokens. Index Synths allow for exposure to particular segments of the industry and diversification of risks without holding and managing various tokens.

sCEX

sCEX is an Index Synth designed to give traders exposure to a basket of Centralized Exchange (CEX) tokens roughly approximating their weighted market capitalization. The current sCEX index consists of Binance Coin (BNB), Crypto.com (CRO), Bitfinex's LEO Token (LEO), Huobi Token (HT), OKEx Token (OKB), FTX Token (FTT), and KuCoin Shares (KCS).

There is also the Inverse Synth called iCEX, which is an inverse of the sCEX Index Synth and works like other Inverse Synths.

sDEFI

With the growing interest in DeFi, the sDEFI Index Synth was introduced to provide traders with an index exposure to a basket of DeFi utility tokens in the ecosystem. The current sDEFI index consists of the following tokens: Aave (AAVE), Synthetix Network Token (SNX), Uniswap (UNI), Maker (MKR), Balancer (BAL), Compound (COMP), Curve (CRV), Kyber Network (KNC), Ren (REN), Sushiswap (SUSHI), UMA (UMA), Yearn Finance (YFI), 1inch (1INCH) and Bancor (BNT).

The inverse of this Index Synth is called iDEFI.

sFTSE and sNIKKEI, in the meantime, track the price of the FTSE 100 Index (FTSE100) and Nikkei 225 Index (NIKKEI225) through price feed supplied by Chainlink oracle.

Synthetix Exchange

Synthetix Exchange is a decentralized exchange platform designed for the trading of SNX and Synths. Synthetix Exchange does not have order books, nor does it have liquidity pools like Uniswap. Synthetix Exchange allows users to trade directly against a smart contract that maintains constantly adequate liquidity, thus theoretically reducing risks of slippage or lack of liquidity.

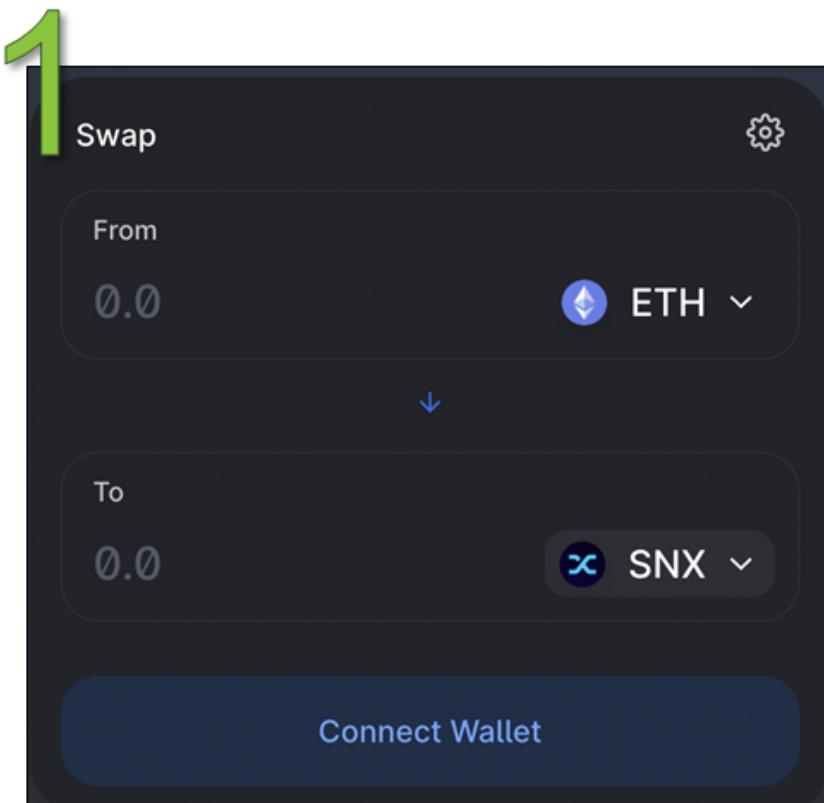
Since users are purchasing a synthetic contract rather than trading the underlying asset, users can buy up to the total amount of collateral in the system without affecting the contract's price. For example, a \$10,000,000 BTC buy/sell order would likely result in considerable slippage in traditional

exchanges, but not in Synthetix Exchange as users trade against the Synthetix contract directly.

For the stats of the Synthetix protocol, please refer to <https://dashboard.synthetix.io/#synths>

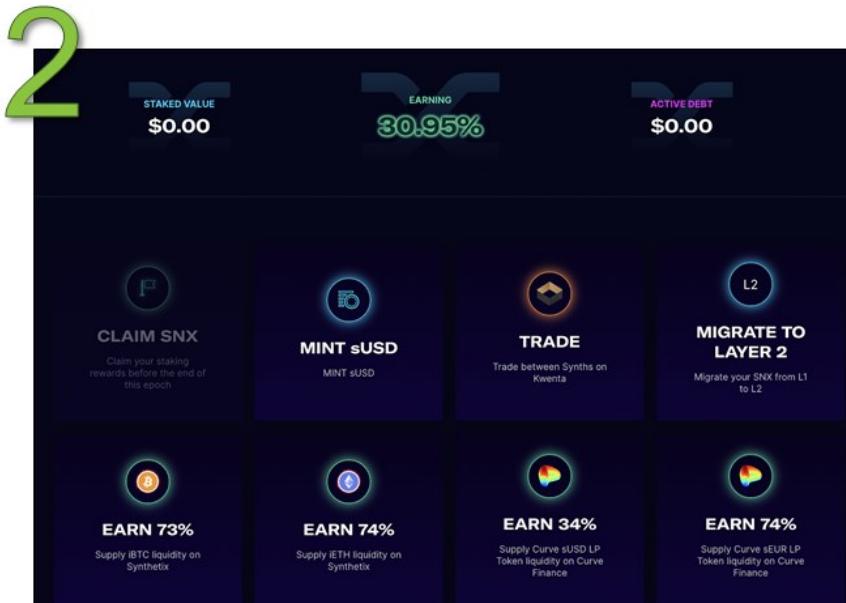
And that's it for Synthetix - if you are keen to get started or test it out, we have included a step-by-step guide on how to mint a Synth. Otherwise, head on to the next section to read more on the next DeFi Dapp!

Synthetix: Step-by-Step Guide



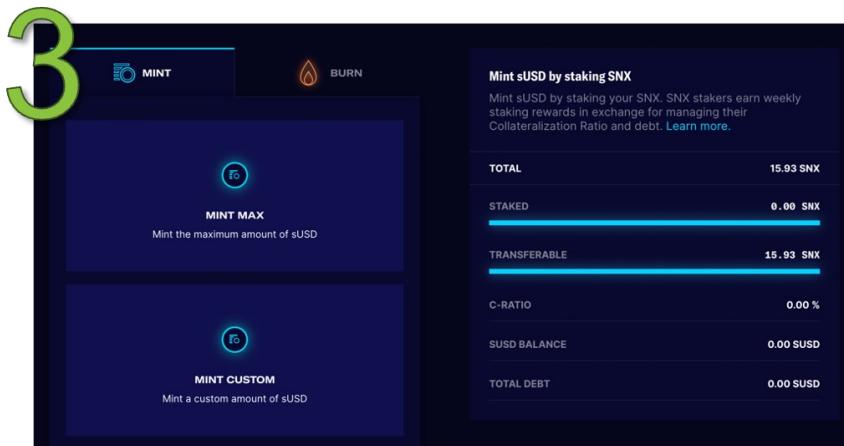
Step 1

- Before you could mint any synth, you will need SNX token to be used as collateral
- If you do not have one, you could check out our [SNX page](#) to see the list of available exchanges to trade for it
- In this tutorial, we swap our ETH for SNX on Uniswap (<https://uniswap.exchange/swap>)
- Connect your wallet and enter the amount of ETH you wish to swap for SNX



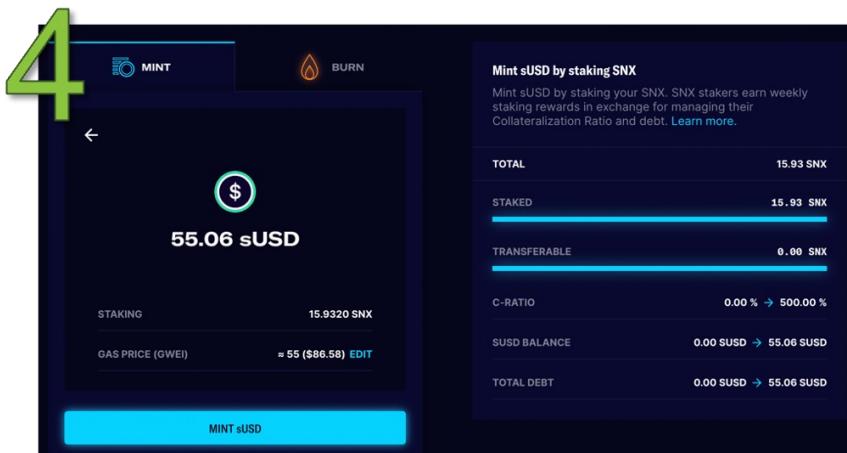
Step 2

- To mint your Synth, go to <https://staking.synthetix.io/>
- Connect your wallet
- Click “MINT sUSD”



Step 3

- Choose “MINT MAX” or “MINT CUSTOM”
- We choose “MINT MAX” for this example



Step 4

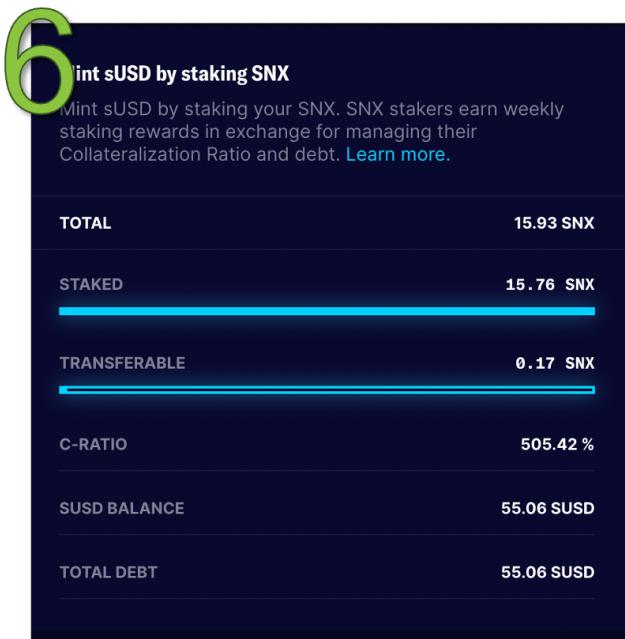
- The available amount to mint depends on the collateral ratio of SNX to the Synth
- As of April 2021, the current collateral ratio is 500%, as shown by the C-RATIO on the right
- So $\$275.55 / 500\%$ is roughly equal to 55.06 sUSD
- Click “MINT sUSD”

Decentralized Derivatives



Step 5

- A Metamask pop-up will show up, then click ‘Confirm’.
- The website will show minting in progress while we are waiting for the transaction to go through.



Step 6

- After confirmation, you will be able to see sUSD in your wallet balance.

Recommended Readings

1. Crypto Derivatives, Lending, and a touch of Stablecoin (Gary Basin) <https://blockgeeks.com/guides/defi-use-cases-the-best-examples-of-decentralised-finance/# Tool 2 DeFi Derivatives>
2. DeFi Use cases: The Best Examples of Decentralised Finance (Rajarshi Mitra) <https://hackernoon.com/crypto-derivatives-lending-and-a-touch-of-stablecoin-59e727510024>
3. The Ultimate Guide To Synthetix. (DefiZap and @DegenSpartan) <https://defitutorials.substack.com/p/the-ultimate-guide-to-synthetix>
4. Synthetix (Cooper Turley and Lucas Campbell) <https://fitznerblockchain.consulting/synthetix/>
5. Synthetix for dummies (TwiceCrypto) <https://medium.com/@TwiceCrypto/synthetix-for-dummies-477a0760d335>
6. Synthetic Instruments In DeFi : Synthetix (Joel John) <https://www.decentralised.co/understanding-synthetix/amp/?>
7. Synthetic Assets in DeFi: Use Cases & Opportunities (Dmitriy Berenzon) <https://medium.com/zenith-ventures/synthetic-assets-in-defi-use-cases-opportunities-19b11f57a776>
8. The Value and Risk of Synthetix (Gavin Low) <https://medium.com/the-spartan-group/the-value-and-risk-of-synthetix-45204346ce>

Opyn



What is Opyn?

Opyn provides protection against price volatility on your assets as well as insurance for smart contracts. Users can get protection for Ethereum (ETH), Wrapped Bitcoin (WBTC), Yearn Finance (YFI), Uniswap (UNI), DeFiPulseIndex (DPI), and USDC and DAI deposits on Compound (COMP).

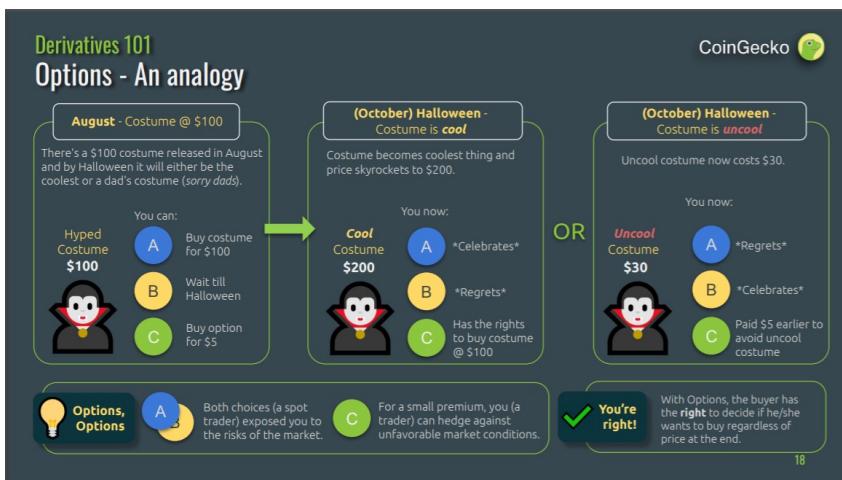
In addition to smart contract failures, Opyn also protects against other risks such as financial and admin risks. Opyn does this by making use of financial derivatives, namely options.

What are Options?

There are two kinds of options: Call and Put. A Call option is a right, but not the obligation to purchase an asset at a specific strike price within a particular period of time. On the other hand, a Put option is a right, but not the obligation to sell an asset at a specific strike price within a particular period of time.

For every purchaser of an option, there must be a seller of an option. A purchaser of an option will pay a premium to the seller of the option to get this right.

Here is a Halloween analogy of a Call option to better help your understanding of options:



There are two main options flavors, namely American and European options. The difference between the two is that for an American option, the buyer can exercise the option anytime before the expiry date, whereas for a European option, the buyer can only exercise it only on the strike date.

How does Opyn work?

Opyn allows users to hedge against the risk of price fluctuations, smart contract exploits, administration/governance risks, and black swan events. For instance, Opyn allows users to buy put options on ETH and WBTC.

By using Opyn, a trader can buy oTokens which can be used as a right to sell ETH and get back USDC in the event of a massive price plunge, thereby receiving protection against downside risk.

For example, let's say that a user purchases one put option on ETH with a strike price of \$2,400. If the price of ETH were to plunge below the strike price, let's say to \$2,000. With Opyn's oToken put options, the option holder can redeem a cash settlement on the price difference, or \$400. Although a premium is charged on purchasing options, which may vary based on market participants, in extreme events, the benefits would far outweigh the cost by limiting the amount of loss incurred by the holder. No centralized entity is needed to be the counterparty for exercising the options, making this a truly decentralized insurance platform.

How much do options cost?

The price of options and insurance on Opyn varies by protocol, option type, cover period, and cover amount. In Opyn version 1, the oTokens minted on Opyn can be traded on decentralized exchanges like Uniswap, but in Opyn version 2, options are traded using an order book model, where buyers and sellers can place their limit orders. Regardless, the value of the oTokens would fluctuate based on the supply and demand of options for a given protocol.

The price quoted usually reflects the intrinsic value of the option. For instance, if the strike price of a put option is \$2,000, whereas the asset's current price is \$1,000, you would expect the price of the put option to be at least the difference between the asset price and the strike price. In this case, the option would be worth at least \$1,000. However, other factors such as time decay or fundamentally negative changes in an asset's value would also be incurred as a discount to the option premium.

One thing to note is that because the pricing of oTokens is determined by supply and demand, one can use this as a signaling mechanism to check if the options are over or undervalued. If people believe that the option is undervalued, they will start purchasing more oTokens, and the oTokens would increase in price.

Why would anyone sell protection on Opyn?

For every purchaser of insurance (purchaser of Put option) on Opyn, there must be a provider of insurance (seller of Put option) on Opyn. By being an insurance provider on Opyn, individuals can earn a yield on their holdings of ETH, YFI, UNI, WBTC, or DPI.

To do so, one needs to start by supplying any of these assets as collateral to Opyn's smart contract. Depending on the insurance sold, different collateralization ratios are required. Compound deposits require a 140% ratio, whereas all other put and call options need 100%.

Once collateral has been provided, oTokens can be minted. From there, premiums can be earned in two ways:

1. Being a Liquidity Provider on Uniswap

As a Liquidity Provider on Uniswap, one can earn transaction fees from individuals buying and selling on the Opyn platform through Uniswap. Liquidity Providers have the opportunity to make a large but variable return from providing liquidity on Uniswap. Liquidity Providers are allowed to remove funds at any time. Our section on Uniswap shows you the steps to provide liquidity on Uniswap.

2. Selling oTokens on Uniswap

The oTokens that have been minted can be sold on Uniswap. To calculate the return for selling oTokens on Uniswap, you can look at Opyn's 'sell protection' tab. Returns for insurance on Compound deposits are the difference between the uninsured yield and insured yield. Returns for call and put options for other assets can be viewed from the 'Earn Premiums' and 'Earn (Annualized ROI)' columns.

The premiums that can be earned on the collateral provided are quite lucrative. However, earning this yield does not come without risk. By selling a put option in return for yield, the option seller assumes the risk that an adverse event will not occur (e.g., technical risks like hacks, financial risks like Dai breaking its peg, or a run on Compound). The individual must also maintain their respective collateralization ratios to avoid liquidation.

Is Opyn safe?

Opyn has a publicly verifiable smart contract and has been audited by OpenZeppelin, a smart contract auditing firm. The full report is available here: <https://blog.openzeppelin.com/opyn-contracts-audit/>

Opyn is also non-custodial and trustless, with a reliance on incentives for it to work.

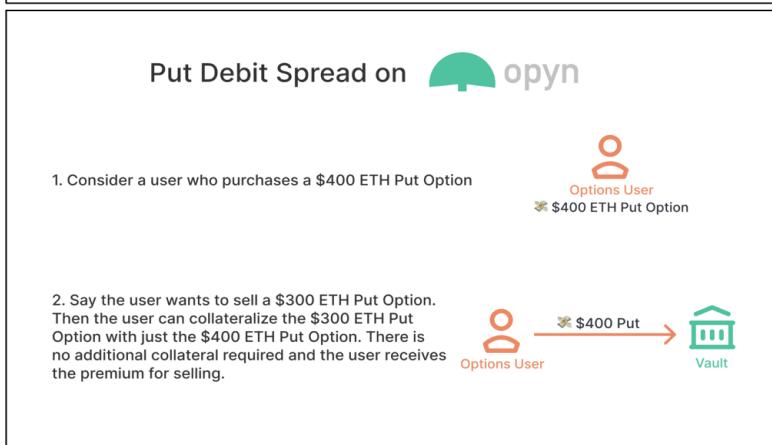
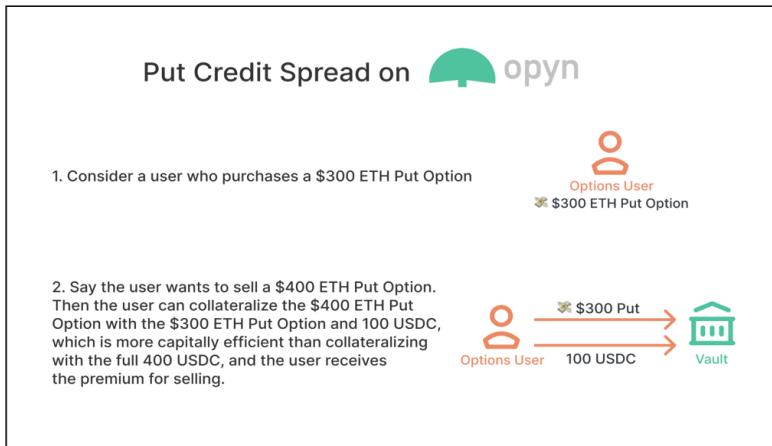
Opyn Version 2

Version 2 of Opyn was released on 30 December 2020 and is called the Gamma Protocol (Version 1 was called the Convexity Protocol). Version 2 offers European cash-settled options as opposed to physically-settled American options in version 1.

Besides the variation in the product, the latest iteration comes with a slew of additional features:

- Margin improvements

Users are allowed to create Put Credit Spreads and Put Debit Spreads, significantly improving capital efficiency



- In-the-money options are automatically exercised upon expiry
- Anyone can create new options if the asset has been whitelisted
- Yield-bearing assets (e.g., cTokens, aTokens, yTokens) can be used as collateral

Version 2 options are traded on 0x Exchange. Although Opyn version 2 has been released, version 1 is still operational. At the time of writing (April 2021), version 2 only supports options on Wrapped Ether.

Now, let's take a look at how you can purchase some put options for yourself through both versions of Opyn.

Opyn Version 1: Step-by-Step Guide

The screenshot shows the Opyn Version 1 web interface. At the top, a large green button with the text "Buy Protection" is highlighted with a yellow border. Above this button, the title "Securing Decentralized Finance" is displayed. Below the title, a sub-headline reads "Opyn allows you to protect your DeFi deposits and hedge ETH risk." There are two buttons: "Buy Protection" (highlighted) and "Sell Protection". The main content area is titled "Hedge ETH Risk" and contains a table of option contracts. The table columns are: Expiry, Strike Price, Type, Current WETH Price, and Protection Cost. The rows show the following data:

Expiry	Strike Price	Type	Current WETH Price	Protection Cost
Dec 25th 2020 16:00 GMT+8	360 USDC	Put	\$598.56	\$1.5468 / 1 WETH
Dec 25th 2020 16:00 GMT+8	400 USDC	Put	\$598.56	\$1.0988 / 1 WETH
Dec 25th 2020 16:00 GMT+8	520 USDC	Put	\$598.56	\$4.6344 / 1 WETH
Dec 25th 2020 16:00 GMT+8	550 USDC	Put	\$598.56	\$4.7551 / 1 WETH
Jan 1st 2021 16:00 GMT+8	500 USDC	Put	\$598.56	\$5.8926 / 1 WETH

Each row has "Buy" and "Sell" buttons next to the protection cost.

Step 1

- Go to <https://v1.opyn.co/#/> and click 'Buy Protection'. We will be insuring some WETH

2

◆ Buy Protective ETH Put Option
Hedge yourself against ETH decreasing in value.

Protected WETH	Expiry	Strike Price *	Current WETH Price	Protection Cost
0 WETH	Dec 25th 2020 16:00 GMT+8	360 USDC	\$596.51	\$1.5307 / 1 WETH >
0 WETH	Dec 25th 2020 16:00 GMT+8	400 USDC	\$596.51	\$1.0873 / 1 WETH >
0 WETH	Dec 25th 2020 16:00 GMT+8	520 USDC	\$596.51	\$4.5862 / 1 WETH >
0 WETH	Dec 25th 2020 16:00 GMT+8	550 USDC	\$596.51	\$4.7056 / 1 WETH >
0 WETH	Jan 1st 2021 16:00 GMT+8	500 USDC	\$596.51	\$5.8312 / 1 WETH >

Step 2

- The ‘Buy Protection’ page lists all the assets Oryn provides options for
- Scroll to the ETH Put Options list and click ‘Buy’ on a Put option of your choosing

3

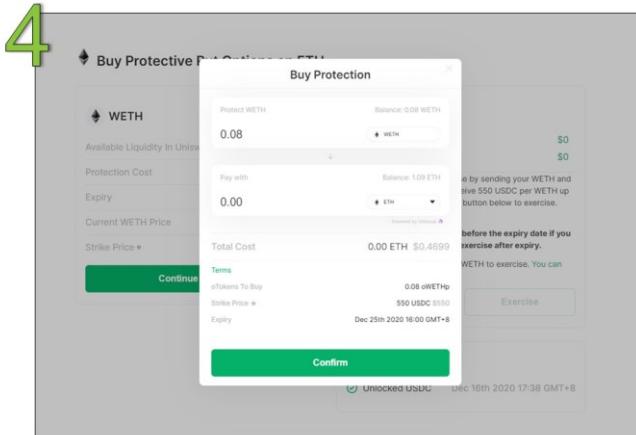
◆ Buy Protective Put Options on ETH

◆ WETH	Sell or Exercise
Available Liquidity In Uniswap	Current Payoffs *
164 oWETHP	Sell early \$0
Protection Cost	Exercise \$0
\$5.95 / 1 WETH	
Expiry	If WETH is below \$550, exercise by sending your WETH and oWETHp to the protocol to receive 550 USDC per WETH up until expiry. You must click the button below to exercise. Learn more about exercising.
12/25/2020	
Current WETH Price	WARNING! you must exercise before the expiry date if you want to exercise. You cannot exercise after expiry.
\$596.47	You must convert your ETH to WETH to exercise. You can wrap ETH here.
Strike Price *	Sell Early Exercise
550 USDC	
Continue Purchase	

Step 3

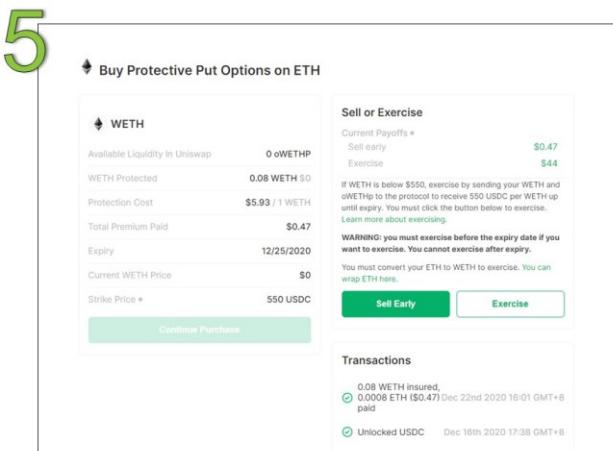
- Selecting ‘Buy’ for any one of the Put options will provide more details
- If the ‘Continue Purchase’ button is greyed out, you may need to wrap your ETH using a DEX
- Click ‘Continue Purchase’ to proceed

Decentralized Derivatives



Step 4

- A dialog box will appear. Set the amount of WETH to cover, as well as which asset you'd like the charge to be denominated in
- Check the details once more, and click 'Confirm' to approve the transaction



Step 5

- Once the transaction has been processed, Opyn will display the transaction.
- You are now insured against an ETH price drop!

Opyn Version 2: Step-by-Step Guide

1

Trade Options on Ethereum

Opyn allows you to earn a premium or hedge risk by trading DeFi options on ETH and ERC20s

START TRADING

Step 1

- Go to <https://opyn.co/#/> and click 'START TRADING'. We will be insuring some WETH.

2

CALLS ↑				PUTS ↓				
IV	Size	Breakeven	Price	Strike	IV	Size	Breakeven	Price
▼	-	-	<input type="text"/>	\$1280	▼	232.47%	0.00 / 20.18	1271.51 <input type="checkbox"/>
▼	187.56%	375.87 / 10.53	2292.35 <input checked="" type="checkbox"/>	\$1760	▼	-	-	<input type="checkbox"/>
▼	154.66%	376.41 / 13.59	2304.30 <input checked="" type="checkbox"/>	\$1920	▼	119.14%	560.88 / 42.12	1889.25 <input checked="" type="checkbox"/>
▼	102.67%	375.00 / 7.80	2467.53 <input checked="" type="checkbox"/>	\$2400	▼	70.32%	794.40 / 32.40	2212.59 <input checked="" type="checkbox"/>
▼	106.09%	449.21 / 225.79	2595.06 <input checked="" type="checkbox"/>	\$2560	▼	-	-	<input type="checkbox"/>
▼	25.00%	0.00 / 0.00	2720.00 <input checked="" type="checkbox"/>	\$2720	▼	-	-	<input type="checkbox"/>
▼	122.95%	847.20 / 502.80	2893.34 <input checked="" type="checkbox"/>	\$2880	▼	-	-	<input type="checkbox"/>

Step 2

- Choose the intended expiry date. In this example, we chose 30-Apr-2021.
- Choose the intended strike price. We chose \$1,280.
- Click on the price

Step 3: Purchase Screen

WEETH 4/30 Put
\$1280 Strike

Step 4: Confirmation Screen

WEETH 4/30 Put
\$1280 Strike

Step 3 Details:

- Position Size: 0.08 oTokens
- Market Impact: 0.0%
- Ox Protocol Fee: 0.00784 ETH
- Step: 1 / 2
- Approve USDC button

Step 4 Details:

- Market Impact: 0.0%
- Ox Protocol Fee: 0.00945 ETH
- Note: Do not change the gas price to buy, otherwise the transaction will fail.
- Buy OTOKEN button

Step 3

- A new window will be shown with more details available, such as the total cost and fee breakdown.
- Key in the amount of oTokens that you would like to purchase.
- Click on the button to approve the spending of your USDC and pay the transaction fee.

Step 4

- Click 'Buy oToken'.
- You're now insured against an ETH price drop!
- If the price of ETH is below \$1,280 on 30th April 2021, we can choose to exercise our oTokens.

Recommended Readings

1. Convexity Protocol Announcement (Zubin Koticha)
<https://twitter.com/snarkyzk/status/1194442219530280960>
2. Options Protocol Brings ‘Insurance’ to DeFi Deposits on Compound (Brady Dale) <https://www.coindesk.com/options-protocol-brings-insurance-to-defi-deposits-on-compound>
3. Getting Started (Opyn) <https://opyn.gitbook.io/opyn/>
4. Opyn launches insurance platform to protect DeFi users (Zubin Koticha) <https://medium.com/opyn/opyn-launches-insurance-platform-to-protect-defi-users-fdcabaca7d97>
5. Exploring the Decentralized Insurance Arena That’s Rising on Ethereum (William Peaster)
<https://blockonomi.com/decentralized-insurance-ethereum/>

CHAPTER 9: DECENTRALIZED FUND MANAGEMENT

Fund management is the process of overseeing your assets and managing its cash flow to generate a return on your investments. We have started seeing innovative DeFi teams building ways for users to better manage their funds in a decentralized manner.

Decentralized fund management is conducted without an investment manager. Instead, algorithms help you conduct trades automatically. This allows you to choose the asset management strategy that best suits your financial goals and reduces the fees paid.

To understand how fund management can work in the decentralized ledger, we will introduce you to Token Sets.

TokenSets



TokenSets is a platform that allows crypto users to buy Strategy Enabled Tokens (SET). These tokens have automated asset management strategies that will enable you to easily manage your cryptocurrency portfolio without executing the trading strategy manually. With an automated trading strategy, you will not need to monitor the market 24/7 manually, thus reducing missed opportunities and risks from emotional trading.

Each Set is an ERC20 token consisting of a basket of cryptocurrencies that automatically rebalances its holdings based on the strategy you choose. In other words, SET essentially implements cryptocurrency trading strategies in the form of tokens.

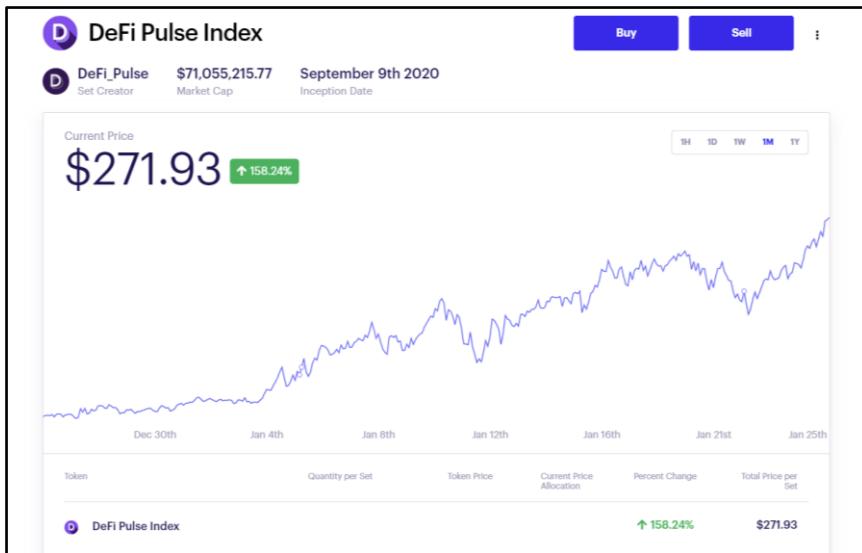
What kinds of Sets are there?

In our first edition, there were Social and Robo Sets. However, it has been phased out in the first quarter of 2021. There are now two kinds of Sets: (i) Index Sets, and (ii) Yield Farming Sets.

Index Sets

Index Sets allow users to have exposure to more assets and reduce gas fees by purchasing a single token instead of buying multiple assets individually. The most popular Index Set is the DeFiPulse Index (DPI), an index that tracks DeFi tokens' performance based on a market capitalization-weighted index.

Decentralized Fund Management



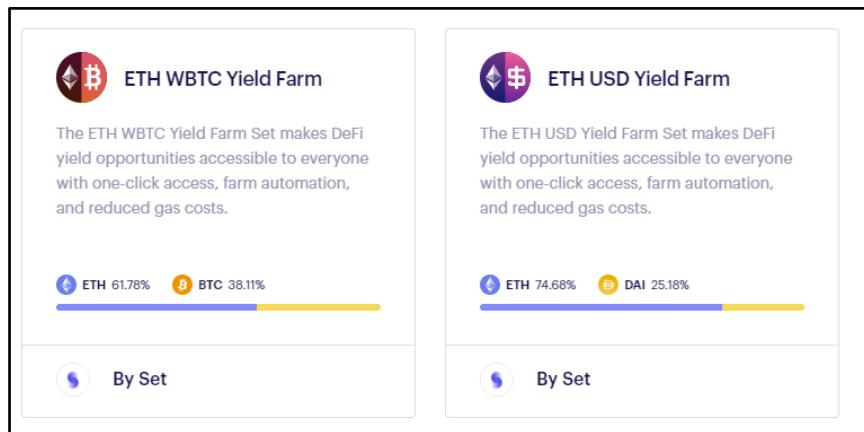
At the time of writing (April 2021), there are 14 DeFi assets under DPI:

1. Uniswap (UNI)
2. Aave (AAVE)
3. Synthetix Network Token (SNX)
4. Sushiswap (SUSHI)
5. Maker (MKR)
6. Compound (COMP)
7. Yearn.Finance (YFI)
8. REN (REN)
9. Loopring (LRC)
10. Balancer (BAL)
11. Kyber Network (KNC)
12. Harvest Finance (FARM)
13. Cream Finance (CREAM)
14. Meta (MTA)

Yield Farming Sets

Yield Farming Sets enable users to save gas by removing the need to constantly conduct multiple smart contract transactions to generate yield on yield farming protocols. These Sets' strategies will periodically claim Liquidity Provider (LP) rewards, sell them for the curated assets, and stake

those assets to generate more LP rewards. Essentially, this is an example of DeFi's take on compounding interest!



How are Sets helpful?

Sets essentially tokenize trading strategies. Suppose you are keen to try out selected trading strategies, such as purchasing a diversified basket of assets. In that case, Set is likely the easiest way to go about doing so.

That being said, always do your due diligence. Just because a Set has historically been performing well does not mean that it will continue to do so. The cryptocurrency market is highly volatile, and the old saying of “past performance is not an indicator of future results” is especially true here. Instead, research and compare the available strategies to see which one makes the most sense to you, and then use TokenSets to get started in no time.

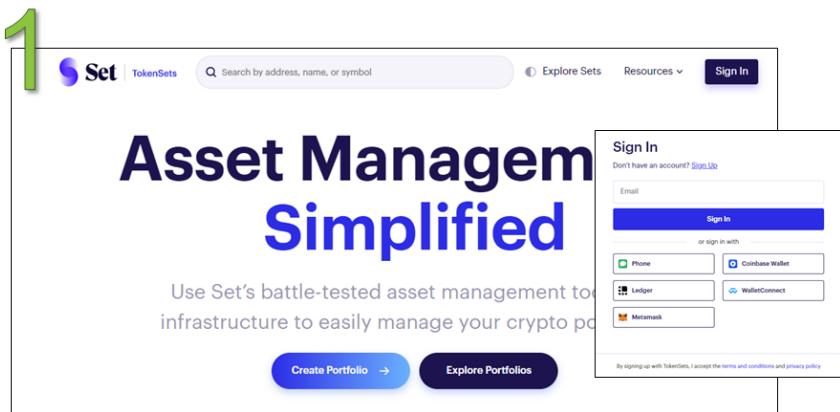
We will be going through one of the popular Index Sets as an example - the DeFi Pulse Index Set. In this case, the Index Set follows the market capitalization weighting strategy, which rebalances the set based on the market capitalization of the underlying assets. This trading strategy saw its value increase by 252% since inception (10 September 2020 – 1 April 2021).

Arguably, this Index Set may not perform as well as some of its underlying assets individually. However, it outperformed most of its underlying assets by a wide margin (11 out of 14). Unless users possess sustainable competitive

advantages on active investing, passive investing through Index Sets is the best option for a diversified exposure in the space.

That's it for TokenSets - if you are keen to get started or test it out, we have included a step-by-step guide on how to get started with Sets. Otherwise, head on to the next section to read more on the next DeFi Dapp!

TokenSets: Step-by-Step Guide



Step 1

- Go to <https://www.tokensets.com/>
- Click “Sign In” on the top right corner
- Choose sign in with Metamask or any wallet of your choice

2

Sets are for Asset Management

A Set is like a digital fund that trades cryptocurrencies to help grow your wealth.



...

Add your email

Get alerts on TokenSet events.
Your email will be hidden from public.

Rebalances
 New Products & Features
 Portfolio Performance Updates

Enter your email

Submit

Not now

Don't show again

Step 2

- It will redirect you to a new interface
- A popup will ask for your email address. You can choose to ignore it as it is optional.

Decentralized Fund Management

The screenshot shows the Set Protocol Explore page. At the top, there's a large green '3' icon followed by the word 'Set'. The navigation bar includes 'TokenSets', a search bar, 'Explore Sets', 'Resources', and 'Account'. Below the header, a section titled 'Explore' says 'Explore portfolios and products created on Set Protocol.' There's another search bar. The 'Featured Sets' section displays three cards:

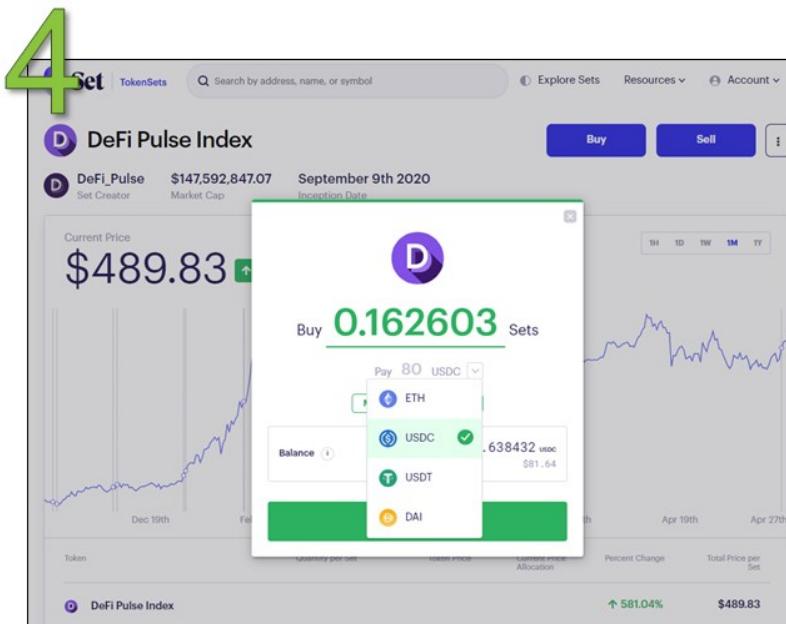
- ETH 2x Flexible Leverage Index**: By DeFi_Pulse. It shows a chart with cETH at 177.86% and USDC at -77.86%.
- DeFi Pulse Index**: By DeFi_Pulse. It shows a chart with UNI at 21.65%, AAVE at 18.3%, and 12 more assets.
- Metaverse Index**: By IndexCoop. It shows a chart with MANA at 17.57%, ENJ at 17.55%, and 13 more assets.

Below this is a section titled 'Explore All Sets' with a table showing market data for various sets. The columns include Name, Market Cap, Price, and performance over 1 Day, 1 Week, 1 Month, 3 Months, and Since Inception. The table lists two sets:

Name	Market Cap	Price	1 Day	1 Week	1 Month	3 Months	Since Inception
ETH 2x Flexible Leverage Index	\$37,432,113.67	\$167.81	↑ +9.7%	↑ +3.7%	↑ +126.3%	—	+59.6%
ETH USD Yield Farm	\$382,484.79	\$391.80	↑ +0.7%	↑ +16.7%	↑ +29.4%	↑ +65.8%	+206.3%

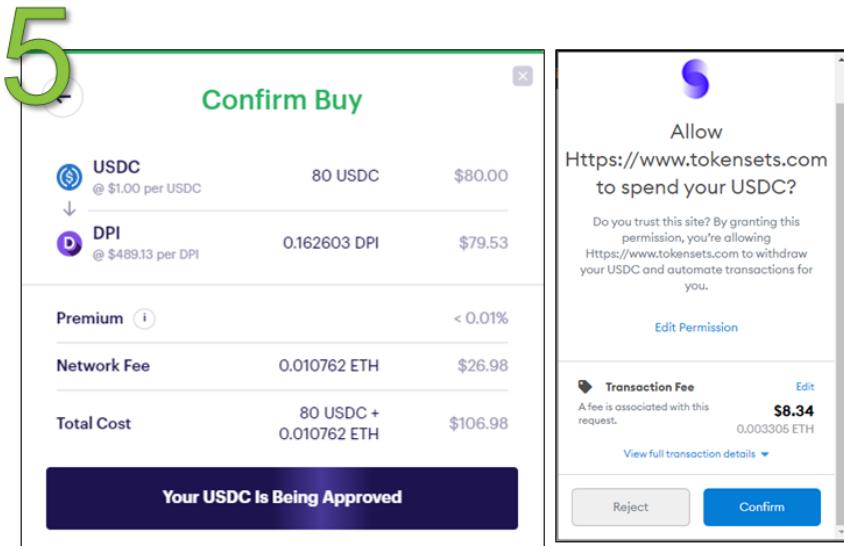
Step 3

- Eventually you will arrive at the Explore page
- You can see a list of Sets that are available.
- Index Sets typically has the word “Index” and Farm Index has the word “Farm”
- For example: “DeFi Pulse Index” and “ETH-USD Yield Farm”



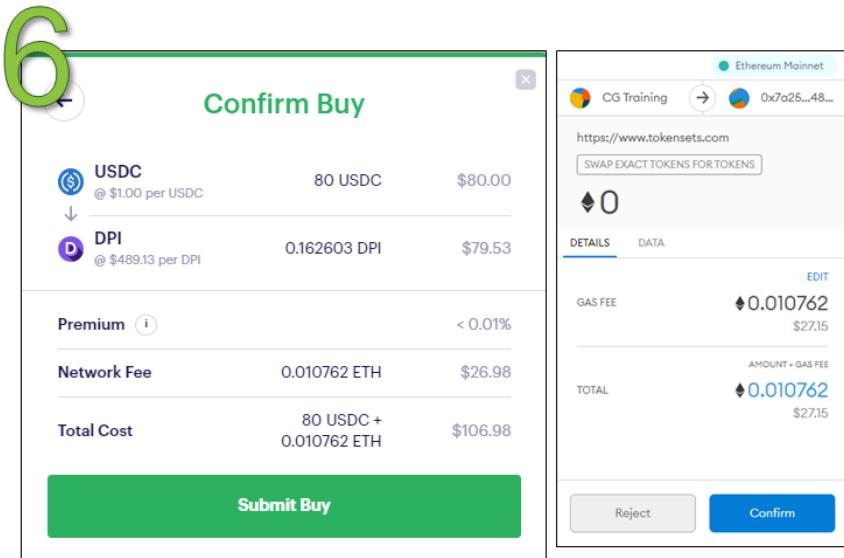
Step 4

- Let's take a look at how to buy one of the Sets. We chose the DeFi Pulse Index as an example.
- Click on DeFi Pulse from Explorer
- Then, click "Buy"
- A pop-up window will show
- You can choose to buy it with one of the four supported currencies
- We chose to buy 80 USDC (\$80) worth of DPI
- Click "Submit"



Step 5

- You will need to approve for TokenSets to spend your USDC
- Click “confirm” on your Metamask



Step 6

- Once approval confirmed, you can now submit your buy
- Another transaction approval is required
- Click “confirm” on your Metamask
- You are done!

Recommended Readings

- Automated Asset Management with Set Protocol. (Zerion)
<https://blog.zerion.io/tokencards-explained-af5771208860?gi=3beb1e590e6f>
- DeFi10 Part 1: Lessons in Building a DeFi Portfolio
<https://thedefiant.substack.com/p/defi10-part-1-lessons-in-building>
- DeFi10 Part2: Becoming a Programmable Money Fund Manager
<https://thedefiant.substack.com/p/defi10-part2-becoming-a-programmable>
- Returns of Hodling versus DeFi-ing (Evgeny Yurtaev)
<https://blog.zerion.io/returns-of-holding-vs-defi-ing-c6f050e89c8e>

CHAPTER 10: DECENTRALIZED LOTTERY

Thus far, we have gone through various protocols for stablecoins, decentralized exchanges, swaps, and derivatives - all of them serious stuff. In this section, we will introduce something light and fun - a decentralized, no-loss lottery.

In [February 2020](#), a user who had deposited only \$10 had won \$1,648 in PoolTogether's weekly DAI Prize Pool, a 1 in 69,738 chance of winning. The best part of PoolTogether's lottery is that participants can get a refund of the \$10 deposit if they did not win. There is no loser in this game, but only opportunity cost involved. Read on to find out more.

PoolTogether



What is PoolTogether?

PoolTogether is a decentralized no-loss lottery or decentralized prize savings application where users get to keep their initial deposit amount after the lottery prize is drawn. Instead of funding the prize money using the lottery tickets purchased, the prize money is funded using the interest earned on Compound protocol by the pooled user deposits. For each round of PoolTogether, all the user deposits are sent to Compound to earn interest. One lucky winner will be selected at random at the end of each interval to win the entire interest accumulated as prize money.

Participating in PoolTogether is relatively straightforward - simply “purchase” PoolTogether tickets using DAI, USDC, UNI, or COMP tokens. Each ticket represents one entry, and the chance of winning increases proportionately with the number of tickets purchased.

PoolTogether currently supports four different lotteries - a weekly DAI, USDC, UNI, and COMP pool. Sponsors can provide additional rewards in the form of any tokens to the pool, and this is known as Loot Box. In terms of security, PoolTogether has gone through several security audits to review their codes.

This concept is not new, and it is similar to Prize-Linked Savings Account (PLSA), where it incentivizes people to save more in their bank’s savings account by providing sweepstakes for lucky winners. PLSA is a popular concept, with banks and credit unions worldwide offering such programs.

One of the best-known PLSA programs is the “Save to Win” program by Michigan Credit Union League.⁴

Why bother with Decentralized Lotteries?

One of the attractions of decentralized lotteries is that funds do not go through intermediaries or brokers but are instead held by audited smart contracts. There is also no lock-up period on funds, meaning that users can withdraw their funds at any moment. Furthermore, the prize draw can be verified on-chain in Ethereum to make sure there is no manipulation.

Traditionally, the protection laws of the gambling industry have made real-world no-loss lottery, such as PLSA programs, restrictive to users from certain geographical areas to join. This is where Decentralized Applications genuinely shine - anyone from anywhere can participate if they have the funds to do so.

What's the Catch?

Surely there can't be free money? Spot on! There's a small catch - the opportunity cost of putting your funds into PoolTogether. If you place your funds into Compound to supply liquidity, you will earn interest from it.

However, if you put your funds into PoolTogether, you will lose the interest earned from Compound. Instead, you now have the opportunity to win the lottery. Effectively, your “fee” to enter the lottery is whatever interest you would have earned by lending it out on Compound.

What are the odds of winning?

The odds of winning depend on the number of tickets purchased. For example, if there are 1,000 tickets in the pool and you buy one ticket, your chance of winning would be 1 in 1,000. You can always check your odds of winning on the PoolTogether account page.

What's new in PoolTogether's Version 3?

Now anyone can create their own no-loss prize pool using PoolTogether with support for any ERC 20 tokens. Plus, the protocol has chosen to

⁴ “What Are Prize-Linked Savings Accounts? - The Balance.” 21 Feb. 2019, <https://www.thebalance.com/what-are-prize-linked-savings-accounts-4587608>.

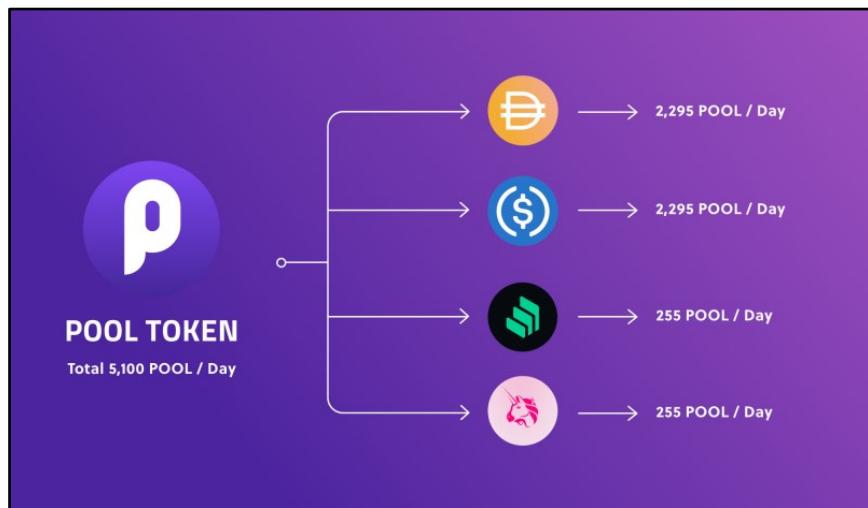
support other yield sources, such as Yearn Finance and Aave, in addition to Compound lending. So now the prize money can be funded using different yield sources.

PoolTogether Token

The governance token for PoolTogether protocol - POOL was released on 18 February 2021. 14% of the total supply was given to all depositors before 14 January 2021. It was a great surprise to the readers of our book that has managed to use the platform.

The airdrop was given based on the amount deposited and how long it has stayed in the pool. For example, one of our colleagues who deposited only 5 DAI for two months was awarded 300 POOL. With an initial trading price of roughly \$20, the airdrop was worth approximately \$6,000.

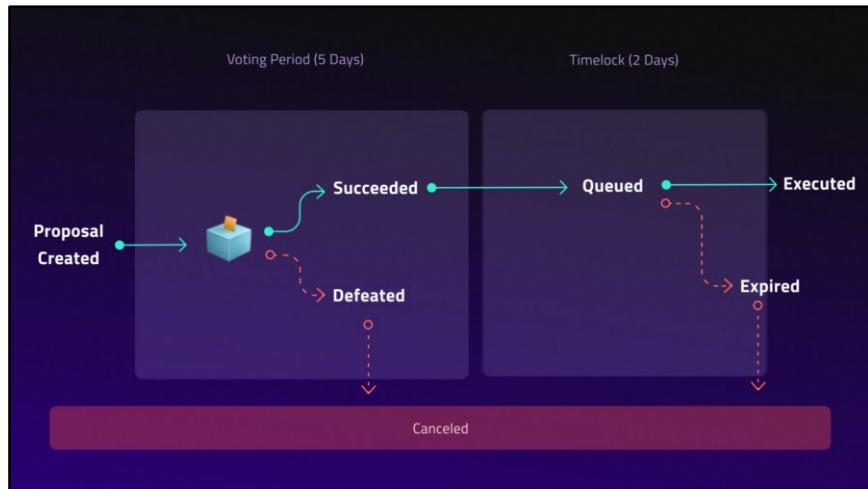
As part of the liquidity mining program, 5% of the total supply will be distributed to all depositors over 14 weeks starting from 17th February 2021. That's a prize pool of 5,100 POOL per day i.e., \$102K per day!



57.54% of the supply is placed under protocol treasury, where the usage will be determined by governance. For example, POOL token holders can vote on:

- Creating a referral program
- Setting up liquidity mining program in the future
- Setting up a grants program

PoolTogether Governance



To submit a governance proposal, users will have to either hold 10,000 POOL tokens (0.1% of total supply) or have 10,000 POOL tokens delegated to them. There is a five-day voting period. With a 1% quorum (at least 100,000 votes are casted), the proposal is considered passed with majority in favor. The proposal will be implemented after a two days timelock.

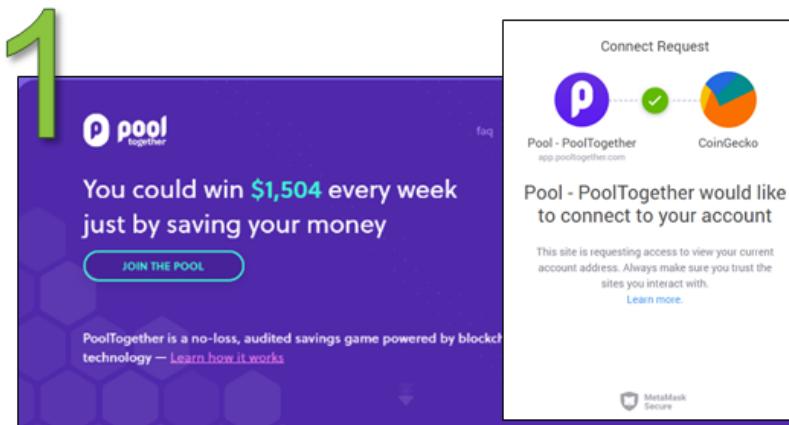
To receive delegation, users will have to delegate themselves on the [Sybil governance website](#).

At first, POOL token holders will be able to vote on:

- Adjust the number of winners for a prize pool
- Adjust the prize frequency
- Approve new prize strategies
- Launch new prize pools

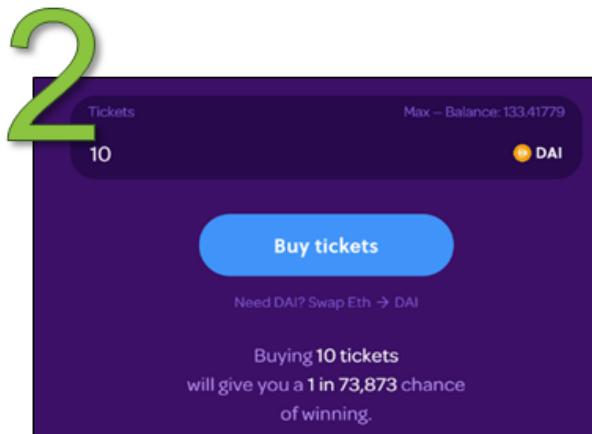
If you're keen to get started or test it out, we have included a step-by-step guide on using PoolTogether. Otherwise, head on to the next section to read more on the next DeFi Dapp.

PoolTogether: Step-by-Step Guide



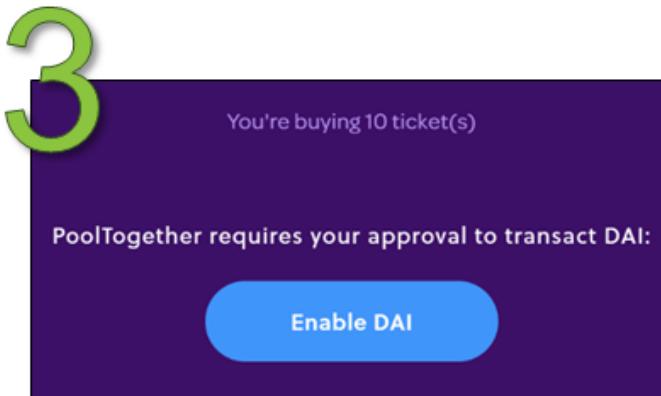
Step 1

- Go to <https://www.pooltogether.com/>
- Connect your wallet Make sure you have DAI, USDC, UNI, or COMP token. We will be using DAI in this example



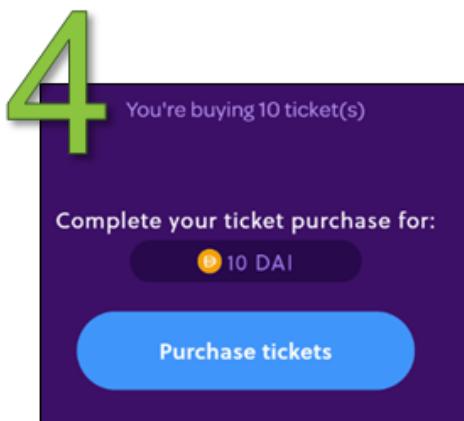
Step 2

- Insert the number of tickets you wish to purchase
- Note: 1 ticket costs 1 DAI and represents 1 entry. Your probability to win goes up with more entries



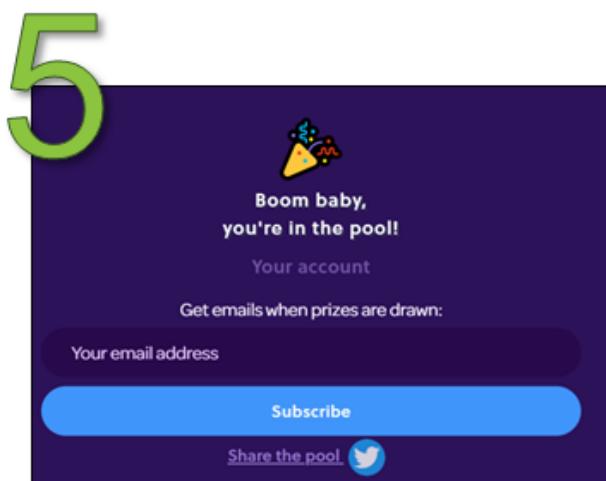
Step 3

- First-time buyer will need to enable DAI



Step 4

- You continue with the purchase afterward



Step 5

- DONE! Just wait for PoolTogether to announce the winner every week

Recommended Readings

1. A Simple Explanation of Risks Using PoolTogether (PoolTogether)
<https://medium.com/pooltogether/a-simple-explanation-of-risks-using-pooltogether-fdf6fec3864>
2. How PoolTogether Selects Winners
<https://medium.com/pooltogether/how-pooltogether-selects-winners-9301f8d76730>
3. No Loss Lottery Now Holds \$1 Million Tokenized Dollars
(TrustNodes) <https://www.trustnodes.com/2020/01/29/no-loss-lottery-now-holds-1-million-tokenized-dollars>
4. PoolTogether - Prize Linked Savings Account (Nick Sawinhy)
<https://defiprime.com/pooltogether>
5. How PoolTogether Turns Saving Money Into a Game (Binance)
<https://www.binance.vision/tutorials/how-pool-together-turns-saving-money-into-a-game>
6. Leighton Cusack Explains How PoolTogether, a No-Loss Lottery Works - Ep. 6 (CoinGecko Podcast)
<https://podcast.coingecko.com/719703/2879608-leighton-cusack-explains-how-pooltogether-a-no-loss-lottery-works-ep-6>
7. A data-driven look inside Pool Together (TokenAnalyst)
<https://research.tokenanalyst.io/a-look-inside-pool-together/>

CHAPTER 11: DECENTRALIZED PAYMENTS

One can already make decentralized payments by sending ETH or DAI directly to the receiver. Now make it better - think cheaper and faster transactions, timed transfers, transfer by conditions, standardized invoicing formats, and more. Some of the more notable projects working on decentralized payments are [Lighting Network](#), [Request Network](#), [xDai](#) and [Sablier](#).

In this chapter, we will be exploring Sablier, a project which we find interesting and can solve some of the outstanding issues for people who are vulnerable in society.

Sablier



What is Sablier?

Sablier is a payment streaming application—meaning that it allows payment and withdrawals to be made in real-time and in small increments (by the second!) between different parties. Think about payments for hourly

consultation work, daily contract workers, or monthly rent payments made in real-time as work and progress are being made. Similar to how you can stream music on Spotify, you can now also stream money on Sablier!

What Does Streaming Payment Mean?

Instead of waiting for a fixed period (e.g., monthly, bi-weekly) for pay, payers can now send payments in real-time in periods defined and agreed upon by both parties. Through Sablier, payees can now receive their payment in real-time and withdraw it whenever they want to.

Why is this important?

We think that Sablier has the potential to help those who live paycheck to paycheck. These people are most vulnerable to delays in their income, where even a few days of delay would mean that they do not have the means to put food on the table.

When that happens, they often resort to payday loans—short-term, uncollateralized loans with very high interest rates (up to 500% APR).⁵ With astronomical interest rates and limited income, payday loan borrowers are especially susceptible to debt spirals—one that has seen many arrested in the US for being unable to repay their loan.⁶

Trust

Streamed payment can be beneficial for new, remote contract workers who, prior to this, had to trust their new employers to pay them for the work they do. When both parties sign a contract through Sablier, they will know that payments are being made, and both parties can verify the payments in real-time.

Timing

Traditionally, salary payments are made monthly or bi-weekly, but there may be instances where funds are required immediately - and payment streaming can help with this. A salaried employee does not have to wait till payday to

⁵ “Payday Loans: Disadvantages & Alternatives - Debt.org.” <https://www.debt.org/credit/payday-lenders/>

⁶ “People are arrested after falling behind on payday loans.” 23 Feb. 2020, <https://www.cnbc.com/2020/02/22/people-are-arrested-after-falling-behind-on-payday-loans.html>. Accessed 24 Feb. 2020.

access his remuneration - he can withdraw as much as he has earned to date, which may resolve immediate concerns. Furthermore, this is also helpful to avoid delays. Even if a worker fully trusts his employer, streaming paychecks guarantees that the payout will be made in full at the end of the period!

Example of how it works

Imagine you provide online consultancy services for a fee of \$60 per hour (\$1 per minute). To begin with, you will likely have to think about whether to:

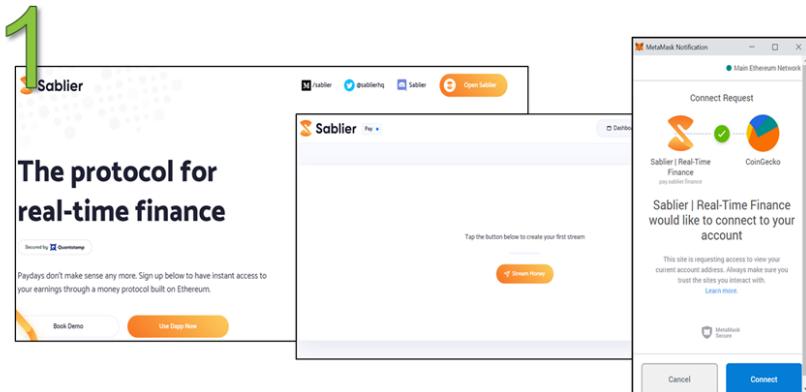
1. Collect payment upfront. However, this may be off-putting to some new clients, or
2. Collect payment later, meaning you will have to trust your client to pay you, or
3. Use an escrow service or platform to protect both sides for a commission.

With the advent of payment streaming, you will no longer need to trust either party. You can be paid on a minute basis to ensure that you and your clients both get their money's worth and that if they do try to run away from paying, you will lose only 1 minute of your time. Essentially, the “trust” part of an online transaction has been shifted from a person to lines of immutable code (the blockchain & smart contract).

This is what Reuben Bramanathan, a cryptocurrency and blockchain consultant, [did to charge](#) his client for a 30-minute consultation.

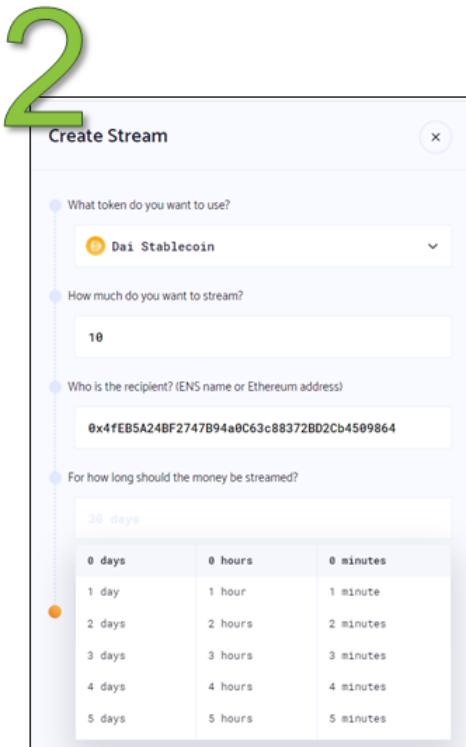
For those curious about making streaming payments, we have included a step-by-step guide on how to get started with Sablier. The process is simple and straightforward.

Sablier: Step-by-Step Guide:



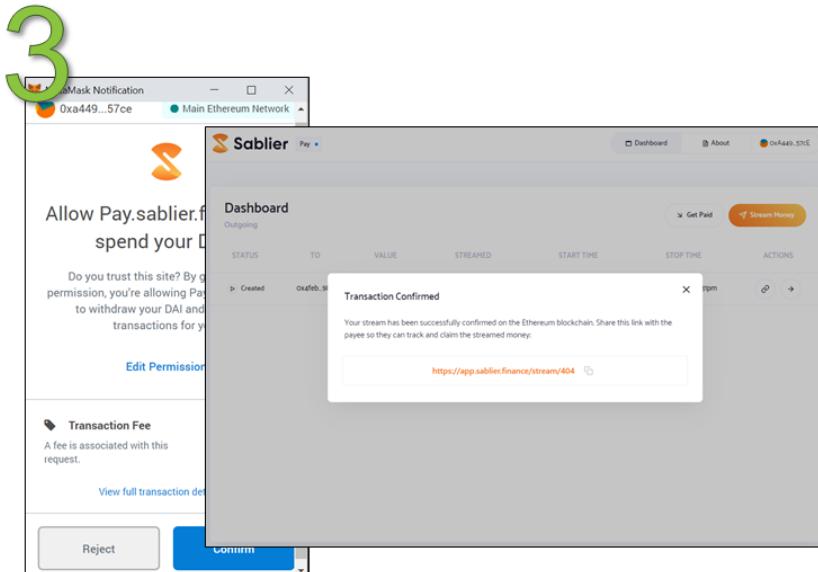
Step 1

- Go to [Sablier](#) and click 'Use Dapp Now'
- You will be redirected to pay.sablier.finance.
- Sign in with your Ethereum wallet



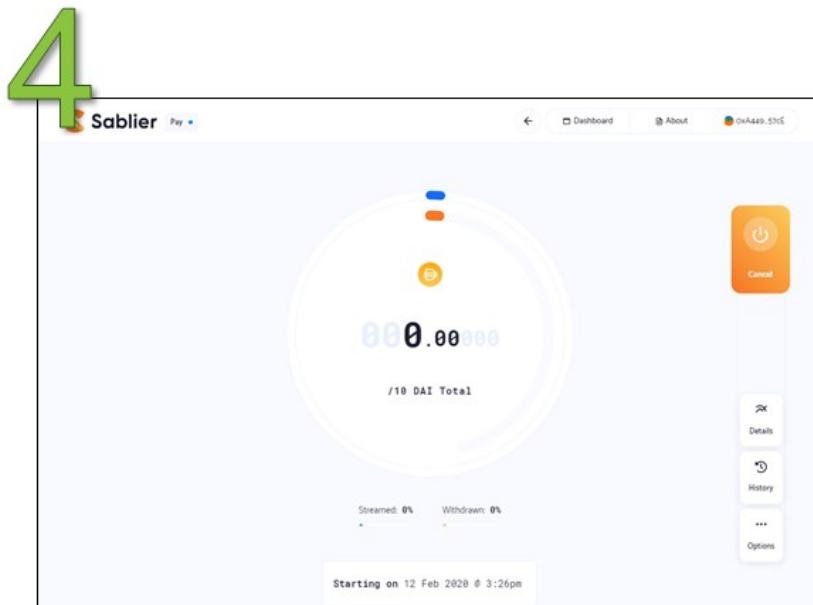
Step 2

- After clicking on ‘Stream Money’, select a token from the dropdown
- Type an amount (which will be refunded if the stream finishes earlier)
- Type an ENS domain or Ethereum address
- Select a duration, e.g. 30 day



Step 3

- Confirm your transaction



Step 4

- After the blockchain validates your transaction, you will be shown a payment link
- Share this with the owner of the ENS domain/Ethereum address from before

Conclusion

Decentralized payment platforms improve the way we pay for services and products, utilizing the power of blockchain technology to deliver real-time payment streams in a ‘trustless’ manner. Such functionality paves the way for new forms of employee-employer and producer-consumer relationships. Furthermore, in today’s environment where contract work agreements are increasingly prevalent and where any shortfall in income could prove detrimental, these platforms can help facilitate better financial and personal outcomes for parties on both ends of a transaction.

Recommended Readings

1. Sablier v1 is Live (Paul Razvan Berg)
<https://medium.com/sablier/sablier-v1-is-live-5a5350db16ae>
2. Sablier The protocol for real-time finance (State of the Dapps)
<https://www.stateoftheDapps.com/Dapps/sablier>
3. Building with Sablier (Sablier)
<https://twitter.com/SablierHQ/status/1214239545220386819?s=19>
4. DeFi Dive: Sablier – the protocol for real-time finance on Ethereum
<https://defipulse.com/blog/defi-dive-sablier-protocol/>

CHAPTER 12: DECENTRALIZED INSURANCE

To participate in DeFi, one has to lock tokens in smart contracts. Tokens locked in smart contracts are potentially vulnerable to smart contract exploits due to the large potential payout. While most projects have gotten their codebases audited, one will never know if the smart contracts are truly safe. There is always a possibility of a hack which may result in a loss.

One of the highest-profile exploits involved a DeFi Dapp known as bZx. The platform suffered two breaches in February 2020 and another in September 2020. The first exploit on 15 February 2020 resulted in a loss of 1,193 ETH (\$318,000) while the second exploit on 18 February 2020 saw 2,388 ETH (\$636,000) lost.⁷ Finally, on 13 September 2020, bZx was hacked the third time, losing \$8.1 million, nearly 30% of its total value locked.⁸ In all, bZx experienced a loss of almost \$10 million in 2020. These breaches involved highly complex transactions involving multiple DeFi Dapps.

More hacks have occurred since then. The fourth quarter of 2020 saw no less than five high-profile attacks against several popular DeFi Dapps. In October 2020, Harvest Finance, a yield farming protocol, was attacked by a hacker who withdrew \$50 million through flash loans. Subsequently, in

⁷ (2020, February 18). Decentralized Lending Protocol bZx Hacked Twice in a Matter Retrieved December 16, 2020, from <https://coinegraph.com/news/decentralized-lending-protocol-bzx-hacked-twice-in-a-matter-of-days>

⁸ (2020, September 14). DeFi Protocol bZx Hacked For Third Time, Loses \$8 Million Retrieved December 16, 2020, from <https://decrypt.co/41718/defi-protocol-bzx-hacked-for-third-time-loses-8-million>

November, Akropolis, Value DeFi, Origin Protocol, and Pickle Finance all suffered varying degrees of security breaches. Altogether, the attacks amounted to a staggering \$69 million loss for the DeFi space. These hacks show that exploits can still occur even though these Dapps had been audited beforehand.

The potential for such massive losses highlights the inherent risks in DeFi and is something that many people do not pay close attention to. Here are some of the risks that DeFi users face:

1. Technical Risks - where smart contracts could be hacked or bugs could be exploited
2. Liquidity Risks - where lending protocols (e.g., Compound) could run out of liquidity
3. Admin Key Risks - where the master private key for the protocol could be compromised

The risks highlight the need for purchasing insurance if one is dealing with large amounts on DeFi. This section will cover two major providers of decentralized insurance to help you protect your DeFi transactions, namely Nexus Mutual and its broker, Armor. Additionally, we also briefly highlight other insurance platforms such as NSure Network and Cover Protocol.

Nexus Mutual



What is Nexus Mutual?

Nexus Mutual is a decentralized insurance protocol built on Ethereum that offers cover for smart contracts on the Ethereum blockchain and custody cover for centralized lenders and exchanges such as Celsius, Blockfi, Nexo, Binance, Coinbase, Kraken, and Gemini. As of December 2020, Nexus

Mutual provides cover for 64 smart contract protocols. Here's a list of some of the more prominent Dapps they cover:

Selected Nexus Mutual Supported DeFi Smart Contracts (Dec 2020)			
No.	DeFi Smart Contract	No.	DeFi Smart Contract
1	MakerDAO	18	Compound
2	Moloch DAO	19	Uniswap v1
3	Nuo	20	Uniswap v2
4	Gnosis	21	Paraswap
5	0x	22	SushiSwap
6	Tornado Cash	23	Yearn Finance
7	Uniswap	24	Cover Protocol
8	Argent	25	Opyn
9	dYdX	26	Celcius
10	Set Protocol	27	Hegic
11	Fulcrum	28	C.R.E.A.M.
12	Aave v1	29	Akropolis Delphi
13	Aave v2	30	Yam Finance
14	Edgeware	31	Bancor Network
15	IDEX	32	Balancer
16	Instadapp	33	Synthetix
17	DDEX	34	Pool Together

What event is covered by Nexus Mutual?

Smart Contract Cover offers coverage against smart contract failures, protecting against potential bugs in smart contract code. The coverage intends to protect against financial losses due to hacks or exploits in the smart contract code. Note that this insurance product only protects against “unintended uses” of smart contracts. Security events that occur through negligence (such as the loss of private keys) are not covered.

Custody Cover aims to protect users who put funds into organizations that hold user funds and assets, such as centralized exchanges and centralized borrowing/lending platforms. Users will be covered in events where:

1. The custodian gets hacked and the user loses more than 10% of their funds, or
2. Withdrawals from the custodian are halted for more than 90 days.

How does coverage work?

Users must first apply to become a member of Nexus Mutual to obtain coverage. Once approved, users can select the Cover Amount and Cover Period for a chosen protocol's smart contract.

The Cover Amount is the amount that users would like to purchase cover for and will be the amount that will be paid out in case there are smart contract failures. The Cover Period is the length of time for which the Cover Amount will be active.

Upon a smart contract failure incident, a Claims Assessment process will take place which Claims Assessors will evaluate. Once the Claims Assessors have approved the Claims Assessment, the user will receive the Cover Amount.

How is the coverage priced?

While all smart contracts can be covered by Nexus Mutual, the price of Smart Contract Cover is based on several criteria such as:

1. Cover Amount
2. Cover Period
3. Value staked by Risk Assessors against the smart contract

For example, let's say you buy 5 ETH worth of cover for the Compound smart contract when ETH is \$200. Assuming the coverage is 0.013 ETH per 1 ETH of coverage for a year, this would cost you a total of 0.065ETH for a year of coverage.

If Compound gets hacked during this period, you will be able to get back 5 ETH regardless of the price of ETH during the time of the hack. If ETH has risen to \$300 during the hack, you would still receive back 5 ETH as long as your claim is approved.

How to Purchase Cover?

1. Specify which smart contract address you want cover for.
2. Specify the Cover Amount, currency (ETH or DAI) and Cover Period.
3. Generate a quote and process the transaction using an Ethereum wallet such as Metamask.
4. You are now covered!

NXM Token

Nexus Mutual has its native token known as NXM. The NXM token is used to buy cover, vote on governance decisions, and participate in Risk and Claims Assessments. It is also used to encourage capital provision and represents ownership to the mutual's capital. As the mutual's capital pool increases, the value of NXM will increase as well.

The token's price is determined using a bonding curve, which is affected by the amount of capital the mutual has and the amount of capital that the mutual needs to meet all claims with a certain probability.

The NXM token is not traded on any exchange and is only used as an internal token for Nexus Mutual. To obtain the NXM token, one must register as a Nexus Mutual member and go through the platform's KYC and AML processes. Users need to pay a one-time membership fee of 0.002 ETH when signing up. Once approved, members can then obtain coverage and enjoy all the benefits that the NXM token provides.

Through a partnership with Armor, users can also buy coverage without going through the KYC processes mentioned above. We will explain further about Armor below.

wNXM Token

wNXM is a 'wrapped' version of the NXM token. In contrast to NXM, wNXM is freely transferable and can be bought on the open market through cryptocurrency exchanges.

To make use of the wNXM token, it first needs to be unwrapped into NXM by a registered Nexus Mutual member. Once unwrapped, wNXM becomes NXM and can be used as usual on the Nexus Mutual platform.

What is a Risk Assessor?

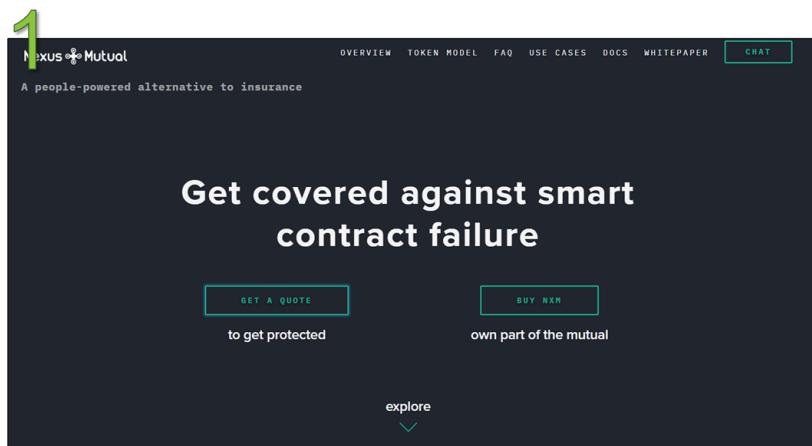
A Risk Assessor is someone who stakes value against smart contracts (essentially vouching that the smart contract is safe). The Risk Assessor is incentivized to do so as the Risk Assessor can earn NXM rewards when users take cover on their staked smart contracts. A Risk Assessor would be someone who understands the risks in Solidity smart contracts and either:

- (1) Assesses individual Dapps themselves, or
- (2) Trusts someone who says the contract is secure (like an auditor or other stakers)

Has NXM ever paid out claims before?

Yes! In the case of the bZx flash loan exploit in February 2020, six members were covered for up to \$87,000. Three claims submitted by these members were accepted for an overall payout of \$34,996. The claims were disbursed immediately after Risk Assessors voted to accept their claims.

Nexus Mutual: Step-by-Step Guide



Step 1

- Go to <https://nexusmutual.io/> and click get a quote

How to DeFi: Beginner

2

The screenshot shows a 'Buy cover' interface with a 'Select project' button at the top left. Below it is a search bar with placeholder text 'e.g. Compound'. The main area displays eight projects in a 2x4 grid:

Project	Type	Cost	Capacity
Cover Protocol	Smart contract	37.79%	156 ETH / 91.1k DAI
SushiSwap	Smart contract	31.09%	274 ETH / 160.5k DAI
Aave V2	Smart contract	7.02%	1.2k ETH / 683.3k DAI
Celsius	Custodian	2.60%	4k ETH / 2.4m DAI
BlockFi	Custodian	2.60%	3.7k ETH / 2.1m DAI
Nexo	Custodian	7.63%	2.5k ETH / 1.5m DAI
inLock	Custodian	34.72%	435 ETH / 254.4k DAI
Lend	Custodian	34.31%	448 ETH / 262k DAI

Each project has a 'Select' button below its details.

Step 2

- Choose a smart contract platform that you would like cover for. We chose Uniswap v2

3

The screenshot shows the 'Buy cover' interface with a 'Get quote' button at the top right. It is divided into two main sections: 'Cover details' on the left and 'Summary' on the right.

Cover details:

- Amount:** 1 ETH
- Period:** 30 DAYS

Summary:

Project	Yearly cost	Capacity
Uniswap V2	2.60%	9389 ETH
		5482928 DAI

A note in the center states: "You're covered for the following events: bugs in the solidity code that lead to a material loss of your funds. Check out full details here."

Step 3

- Fill in the cover amount and period. The minimum cover amount is 1 ETH or 1 DAI, whereas the cover period ranges from 30-365 days
- When done, click 'get quote'

The screenshot shows a step-by-step process for buying insurance. Step 4, titled 'Buy cover', is currently active. The flow consists of four steps: 'Select project' (Step 1), 'Get quote' (Step 2), 'Membership' (Step 3), and 'Confirm' (Step 4). The 'Membership' step is highlighted with a green bar at the top.

Membership Fee

Agreement

- You own part of the mutual**: You can buy cover and earn more NIM by helping run the mutual, including voting on claims, deciding which smart contracts are secure and voting on proposals.
- Legal Entity**: Nexus Mutual is a blockchain-based organisation with a legal structure making sure that the members are not personally liable for the mutual as a whole. To obtain these benefits there are a small set of legal requirements to joining the mutual.
- Membership costs 0.002 ETH (~1.17\$)**: A small membership fee is a legal requirement.
- KYC / AML**: To become a member, you will need to verify your identity using our Know-Your-Customer / Anti-Money-Laundering process. If this fails, your membership fee will be refunded to you.

Summary

	Uniswap V2
Cover amount:	1 ETH
Cover period:	30 days
Quote NIM:	0.0556 NIM
Quote ETH:	0.0021 ETH

Continue

Step 4

- After the quote has been generated, Nexus Mutual will show the cost of coverage. In addition, the platform will prompt you to sign up to be a member
- To become a member, you will need to:
 - Not be a resident in the following countries: China, Mexico, Syria, Ethiopia, North Korea, Trinidad and Tobago, India, Russia, Tunisia, Iran, Serbia, Vanuatu, Iraq, South Korea, Yemen, Japan or Sri Lanka
 - Complete KYC and AML
 - Pay a one-off membership fee of 0.002 ETH
 - Once your membership is approved, you can proceed to buy your insurance

5

Buy cover

Select project Get quote Confirm

Payment options

Quote ETH: Balance: 1.0 ETH
0.0259 ETH

Quote NXM: 0.6774 NXM

Buy cover using ETH NXM

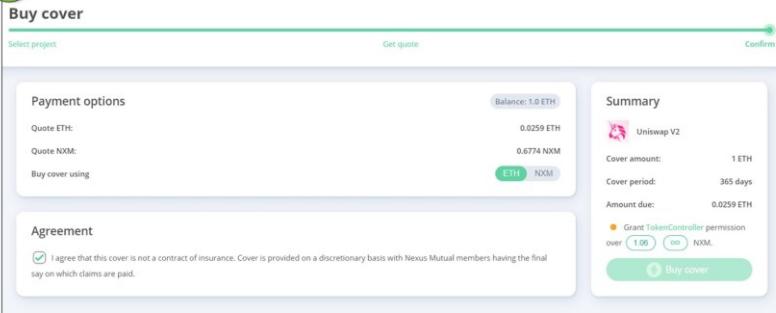
Summary

Uniswap V2

Cover amount: 1 ETH
Cover period: 365 days
Amount due: 0.0259 ETH

Grant TokenController permission over 1.00 ETH over 0.6774 NXM.

I agree that this cover is not a contract of insurance. Cover is provided on a discretionary basis with Nexus Mutual members having the final say on which claims are paid.



Step 5

- Tick the agreement box if all the displayed details are accurate
- Grant permission to Nexus Mutual to connect to your wallet and click 'buy cover'
- You have just bought some insurance!

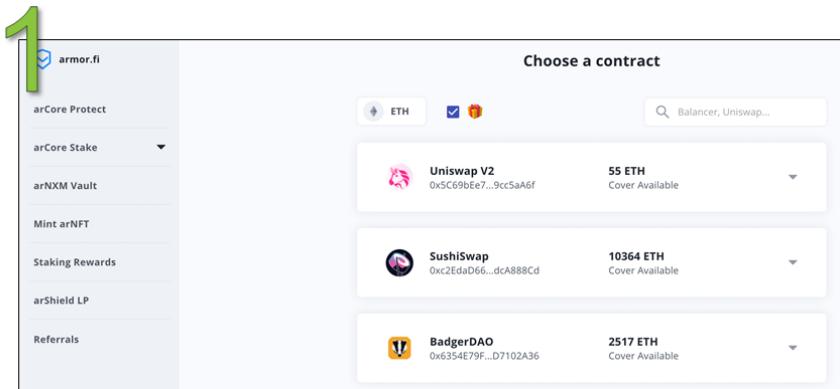
Armor

The screenshot shows the Armor DeFi platform's user interface. At the top, there are social media sharing icons (Twitter, Telegram, LinkedIn) and a 'TOOLTIPS:' button. Below the header is a large blue hexagonal logo with a white 'V' shape inside. A green button at the top left says 'PAY AS YOU GROW'. The main title 'Smart DeFi Asset Coverage' is displayed in bold white text. A subtext below it states: 'Armor is a decentralized brokerage for cover underwritten by Nexus Mutual's blockchain-based insurance alternative'. A 'CONNECT' button with a wallet icon is present. A link to 'Read our Gitbook to learn more about Armor' is also visible. On the left, there's a 'Smart Cover' section with a lock icon and the text: 'arCORE tracks and protects your crypto assets, you pay per second!'. In the center, there are three cards: 'arNFT' (gift icon), 'arNXM Vault' (bar chart icon), and 'Shield Vaults' (shield icon). Each card has a brief description: 'Buy a cover that can be sold, traded or staked for rewards', 'Swap and deposit your (w)NXM tokens and earn yield', and 'Auto-protect your liquidity positions without extra costs' respectively.

Armor is the first insurance aggregator for DeFi. Leveraging the underwriting capability of Nexus Mutual, it offers pay-as-you-go insurance products and the ability to buy insurance covers without KYC.

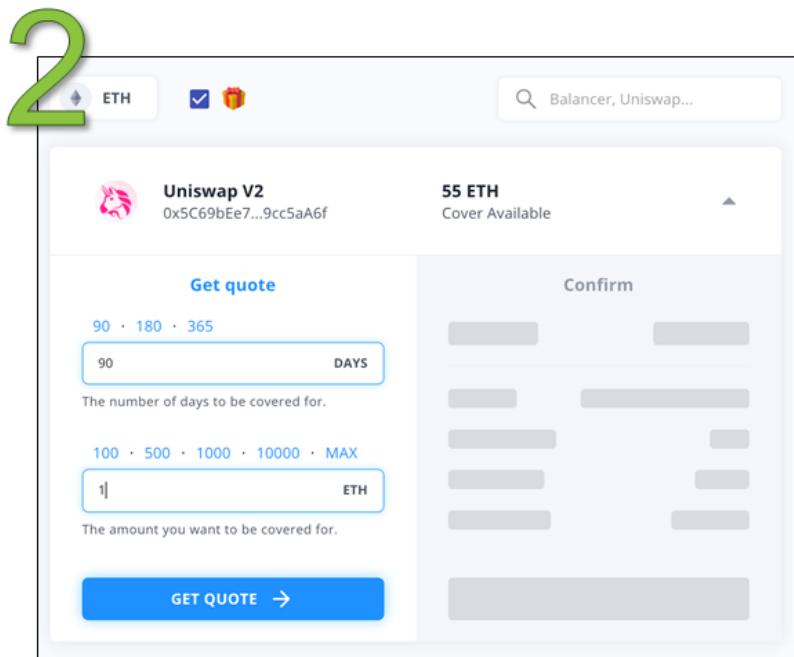
Users who do not want to undergo KYC or cannot purchase cover due to Nexus Mutual's geographical restrictions can obtain coverage through Armor.

Armor: Step-by-Step Guide



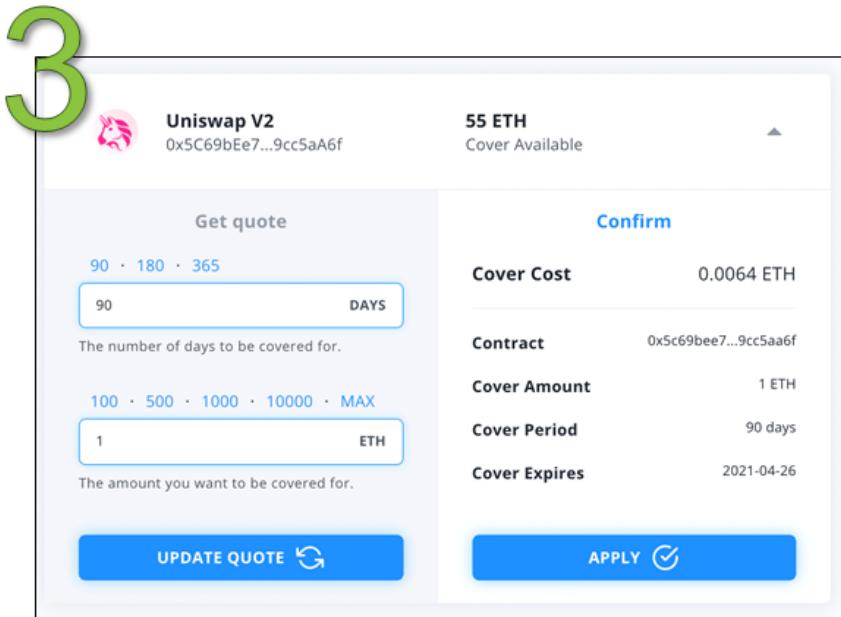
Step 1

- Go to [Armor.fi](#) and click 'mint arNFT'



Step 2

- Choose the smart contract you would like cover for. We chose Uniswap v2 again. Fill in the cover period and amount.
- Click ‘Get Quote’



Step 3

- Armor will generate a quote. Click apply and authorize the transaction through your wallet.
- You have just bought some insurance!

Other Insurance Platforms

As the DeFi space develops and matures further, the complexity of transactions and propensity for Dapps to execute ‘money lego’ interactions increases. Money lego is a concept that describes DeFi architecture that is highly composable and interoperable; Protocols can be built atop each other, leveraging the existing services offered rather than building it out independently. However, high composability can be a double-edged sword as it introduces systemic risk as now a failure can potentially affect more protocols. This natural progression will continue to bring greater decentralization and innovation while at the same time magnifying potential technical, financial, administrative, and smart contract risks.

Addressing these potential risks and protection gaps will remain a challenge for the foreseeable future due to the nascent nature of DeFi. In this section,

we will briefly discuss two more insurance platforms that have recently entered the DeFi insurance space - Nsure Network and Cover Protocol.

Nsure Network



Nsure Network is a decentralized insurance project that borrows the idea of Lloyd's of London, a marketplace to trade insurance risks. Prices for premiums are determined using a Dynamic Pricing Model where capital supply and demand jointly determine the premiums, similar to the free market's pricing mechanism.

Nsure Network's governance tokens (NSURE) are backed by the policies purchased. The price is self-adjustable to the movement of capital supply and demand but is subject to being stabilized by the underlying model. Unlike Nexus Mutual that follows a mutual model, Nsure Network follows a shareholders model where the Nsure Network's tokens are akin to holding a share of the network.

Cover Protocol



Cover Protocol is a peer-to-peer insurance marketplace that operates similarly to a prediction market. Unlike other insurance protocols, the governance token is not used for underwriting risk.

To bootstrap coverage for this new protocol, market makers are incentivized to stake DAI or yDAI collateral to mint CLAIM and NOCLAIM tokens. Cover Protocol insurance covers a selected protocol, and all have a specified expiry date. During the expiry date, either CLAIM or NOCLAIM tokens will have the full claim to the collaterals.

For example, if 100 DAI is used to provide coverage for the Compound protocol until a set expiry date, it will yield 100 CLAIM and 100 NOCLAIM tokens. On the expiry date, if there is a valid claim event, all CLAIM tokens will receive 1 DAI while all NOCLAIM tokens will expire worthless. Conversely, if there is no valid claim event, all NOCLAIM tokens will receive 1 DAI while all CLAIM tokens expire worthless.

To have 100 DAI worth of insurance coverage, users will need to purchase 100 CLAIM tokens from an exchange, as this will expire to 100 DAI in the event of a valid claim. The cost of buying the 100 CLAIM tokens will be the insurance premium.

Conclusion

Insurance is still a very niche segment in DeFi and generally attracts very few retail market participants. However, as more sophisticated players join the market, they will undoubtedly demand better risk management tools.

At the end of the day, the choice to insure or not to insure is ultimately up to you. However, we at CoinGecko would definitely recommend purchasing insurance as anything can happen in DeFi.

Recommended Readings

1. A guide to financial risk in DeFi (Seth Goldfarb)
<https://defiprime.com/risks-in-defi>
2. Nexus Mutual NXM Token Explainer (Hugh Karp)
<https://medium.com/nexus-mutual/nexus-mutual-nxm-token-explainer-b468bc537543>
3. Nexus Mutual (Fitzner Blockchain)
<https://tokentuesdays.substack.com/p/nexus-mutual>
4. The Potential for Bonding Curves and Nexus Mutual (Fitzner Blockchain) <https://tokentuesdays.substack.com/p/the-potential-for-bonding-curves>
5. Why Nexus Mutual should be on your radar (DeFi Dad)
https://twitter.com/DeFi_Dad/status/1227165545608335360?s=0
6. Cover Protocol - Decentralized Insurance Marketplace
<https://defiprime.com/cover-protocol>
7. Armor.fi Living Documentation <https://armorfi.gitbook.io/armor/>
8. Total coverage sold breached \$800 mil only one month after launch
<https://twitter.com/ArmorFi/status/1365872579119108098?s=20>

CHAPTER 13: GOVERNANCE

Most published literature focuses on effective corporate governance for large public companies. However, for DeFi protocols to govern themselves successfully, innovative methods have to be implemented.

To provide some context, imagine managing an online community across the world that consists of members from different time zones, possess various objectives, and are mostly anonymous. This becomes even more complex when large amounts of capital are involved, and communication is largely limited to social media platforms (e.g., Telegram, Discord, etc.)

So how do DeFi projects manage conflicting interests and power struggles? Who gets to decide on the usage of funds and their distribution methods? What initiatives do projects need to prioritize?

Unlike traditional companies with a centralized management structure, DeFi protocols are not required to follow any rules other than those encoded on-chain. This is commonly referred to as “Code is Law”.

DeFi protocols organize themselves as Decentralized Autonomous Organizations (DAOs), organizations governed by smart contracts on the blockchain. DAOs allow groups of people to cooperate without centralized management and coordinate around a shared set of rules to achieve a common mission.

To address DeFi protocol governance, projects such as Compound pioneered the governance token model. Governance tokens provide voting power, allowing token holders to vote on protocol proposals that any community member can submit.

Admittedly, this system is not perfect. There are all kinds of problems such as voter apathy, high participation cost due to gas fees, and non-binding voting results (failure to reach quorum). Nevertheless, it is essential to remember that DeFi is still in its infancy. Until a newer and better governance system is developed, this model will remain the industry standard.

Now that you have a better understanding of how DeFi protocols govern themselves, we would like to share two projects that offer governing tools for DeFi projects to use: Aragon and Snapshot.

Aragon



What is Aragon?

Aragon is a community-driven project with the mission to empower freedom by creating tools for decentralized organizations to thrive. As technology advances at an unprecedented speed, there are concerns about its role in creating a global regime of centralized control, surveillance, and oppression. Aragon is determined to create a society that is free, open, and fair.

Aragon was born to disintermediate the creation and maintenance of companies and other organizational structures.

The Aragon project is stewarded by the Aragon Association, a non-profit entity based in Zug, Switzerland, and governed by Aragon Network Token (ANT) holders.

What is the Aragon Court?

Aragon Court is the solution created by Aragon that mimics the function of the legal courts in the traditional business world. When there are conflicts, participants can escalate them to Aragon Court, where jurors will decide whether any parties should be penalized.

Jurors are randomly selected from the juror pool to review and rule on the dispute by locking ANJ tokens. Unlike traditional courts, jurors are compensated with dispute fees when they vote for the majority result. As such, the system is not catered for unbiased decisions but rather leans on social consensus.

Jurors that voted for the minority result will have a part of their ANJ tokens taken away and rewarded to jurors that voted for the majority. If the result is unsatisfactory, any party can pay a fee to appeal the dispute where a larger number of jurors will review it. This process can be repeated until the entire juror pool rules on the dispute.

Jurors also earn monthly subscription fees on top of the dispute fees when they are not drafted.

How to become a juror?

To become a juror, you will need to have at least 10,000 [Aragon Court \(ANJ\) tokens](#), which can be purchased from DEXs such as Uniswap. ANJ is created by locking [Aragon \(ANT\) tokens](#) in a bonding curve - the more ANT that is locked, the more ANJ that can be created.

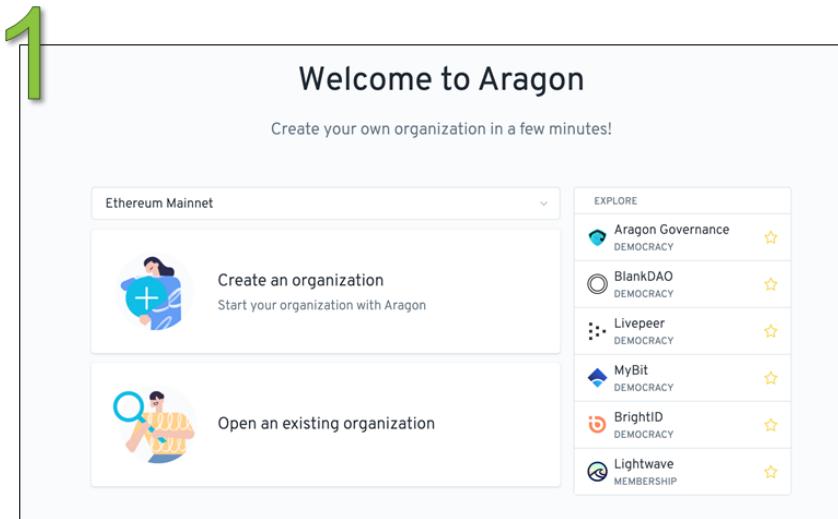
As of April 2021, there is an ongoing merger of ANJ and ANT because the community is skeptical of the need to have two tokens and believes that the complexity outweighs its benefits. The merger is expected to be completed in two parts.

The first part is live as of April 2021, where ANJ holders can redeem their tokens at 0.015 ANT per ANJ. The second part of the ANJ merger will enable ANJ holders to redeem locked-up ANT. The rate will be 0.044 ANT per ANJ, with a 12-month lock.

Who uses Aragon?

Prominent DeFi projects such as Aave and Curve use Aragon. As of September 2020, more than 1,600 DAOs have been created with \$350 million stored in Aragon.⁹

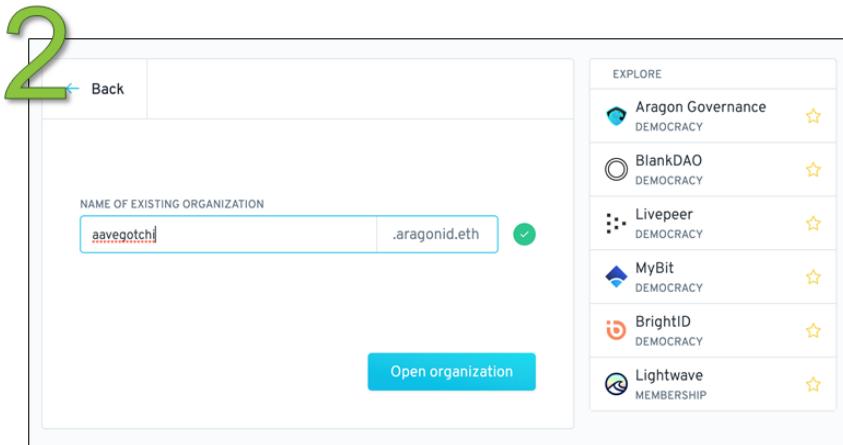
Aragon Step-by-Step Guide



Step 1

- Visit <https://client.aragon.org/#/>, choose “Open an existing organization”.

⁹ Powered by Aragon. Retrieved January 29, 2021 from <https://poweredby.aragon.org/>



Step 2

- In this example, we are going to vote for the Aavegotchi protocol. After typing in the name, we can see the organization is available, as shown with the green tick.
- Click “Open organization”.

Governance



Voting

Open votes

Fundraising

#2: Fundraising: Update tap for DAI with a rate of about...
YES 7%
NO 1%

02D 20H:13M:25S

Fundraising

#1: Fundraising: Update fees deducted from buy and sell orders to respect...
YES 7%
NO 1%

02D 20H:12M:32S

Closed votes

Fundraising

#0: Fundraising: Update tap for DAI with a rate of about...
YES 7%
NO 1%

02D 20H:13M:25S

This screenshot shows the Aragon interface for a Beavegotchi DAO. The 'Voting' tab is selected in the sidebar. It displays three open votes under the 'Fundraising' category. The first vote (#2) is currently active, showing a 'YES' option at 7% and a 'NO' option at 1%. The second vote (#1) has a similar structure but is not yet active. Below these are closed votes, with one entry for vote #0 also under the 'Fundraising' category.

Step 3

- Click the voting tab on the left hand menu. Open votes will be shown.
- Choose the one that you want to vote for.



Vote #2

DESCRIPTION

Fundraising: Update tap for DAI with a rate of about 150000.000000000000000000 DAI per month and a floor of 50000 DAI

CREATED BY

0x8eFe_Ea81

STATUS

Time remaining
02D 20H:13M:10S

SUPPORT %

99.91% (>50% needed)

MINIMUM APPROVAL %

6.81% (>8% needed)

CURRENT VOTES

Yes 100% 1584945.36039 GHST
No 0% 1348.41978 GHST

✓ Yes ✕ No

Voting with 2028 GHST. This was your balance when the vote started (block 11679403, mined at 21:02 on 18th Jan. 2021).

This screenshot shows a detailed view of a single open vote titled 'Vote #2'. The description indicates it's a 'Fundraising' proposal to update the tap for DAI with a rate of about 150,000 DAI per month and a floor of 50,000 DAI. The vote was created by the address 0x8eFe_Ea81. The status bar shows 99.91% support, which is above the required 50% threshold. The minimum approval percentage is set at 6.81%, which is above the required 8% threshold. The current votes show 100% support for 'Yes' with 158,494.536039 GHST and 0% support for 'No' with 1348.41978 GHST. A note at the bottom states that 2028 GHST was the voter's balance when the vote started. The sidebar on the left shows the 'voting' tab is selected.

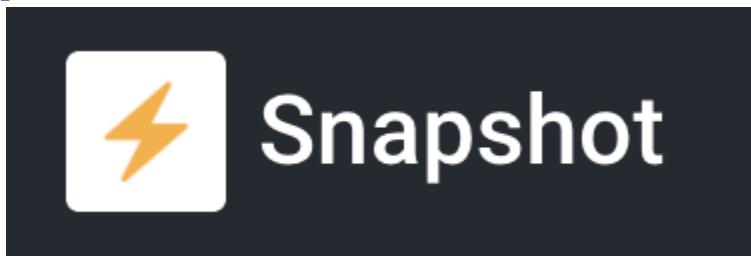
Step 4

- Choose Yes or No and complete the Metamask transaction.
- You have voted!

Recommended Readings

1. Aragon DAOs (Placeholder VC)
<https://www.placeholder.vc/blog/2020/5/7/aragon-daos>
2. ‘The World is Crying for the Tech We’re Building’ An interview with DAO maker Aragon’s Luis Cuende (Decrypt)
<https://decrypt.co/32280/the-world-is-crying-for-the-tech-were-building-an-interview-with-dao-maker-aragons-luis-cuende>
3. DAOs Will Never Govern the World At This Pace (Coindesk)
<https://www.coindesk.com/daos-govern-world-pace>

Snapshot



What is Snapshot?

Due to rising Ethereum gas fees, costs to vote on-chain have become unbearably expensive, hindering small token-holders from participating in the governance process. Snapshot attempts to solve this by taking a snapshot of votes off-chain, effectively making the voting process gas-free.

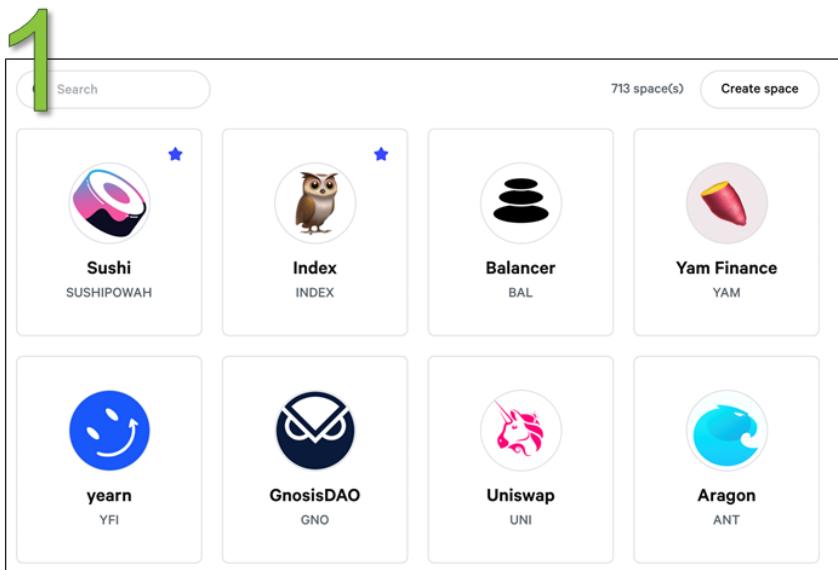
Snapshot has seen many DeFi projects making use of its tool to set up governance processes. As of January 2021, Snapshot is free to use, and as many as 418 different projects have already registered to use Snapshot.

What are the shortcomings of using Snapshot?

Because Snapshot uses off-chain voting, voting results obtained are not binding on-chain by smart contracts. Rather, project teams or multi-sig holders will have to carry out the voting result on-chain, which might not happen if the result is contentious.

Effectively, Snapshot becomes just a poll where the power of execution still lies on other parties. Although this means that power is still somewhat centralized, the solution provided by Snapshot remains a practical and cost-effective solution for DeFi projects to allow community members to participate in the governance process.

Snapshot: Step-by-Step Guide



Step 1

- Visit <https://snapshot.page/#/> and pick the project that you want to vote for.
- For this example we will look at Sushiswap (SUSHI).

All Core Community Active Pending Closed

Active It is suggested to subsidize gas expenses
#QmcJiUj By 0x86C5...E68D end in 7 months

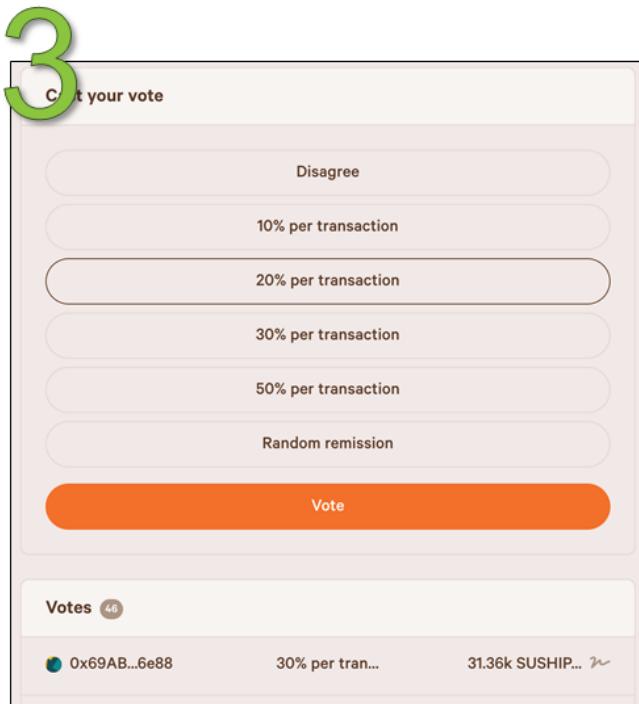
Active Sushi on ZKSwap
#QmYdzGx By 0xBCf7...E09d end in 1 day

Closed Onsen on BSC
#QmTT7KR By 0x4952...2B43 1.49 SUSHIPOWAH ended 4 hours ago

Closed fewdsvaxwes
#QmT4yzm By 0x4472...1694 ended 2 days ago

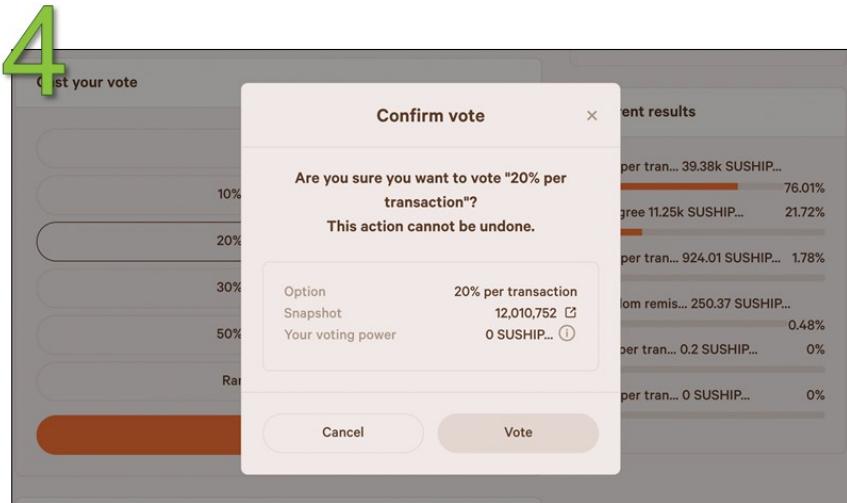
Step 2

- Click the active proposal that you want to vote for. In this example, we are going to choose the first proposal.



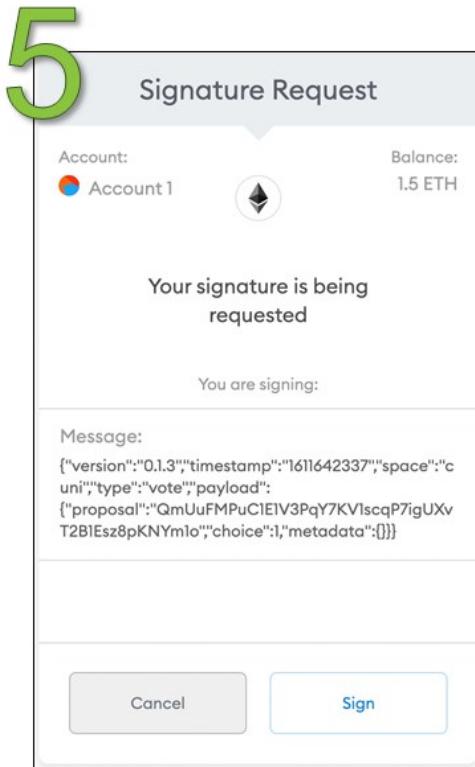
Step 3

- Read the proposal and scroll down until you see the voting options. We can see the voting history of addresses that have voted. In this example, SUSHI token is required for voting.
- Click the option that you want to vote for and click vote.



Step 4

- After verifying the voting power and the correct option, click vote.



Step 5

- Click sign on the signature message. This whole operation will not incur any fees.
- You have voted. You can go back to the proposal page to check your vote history.

Recommended Readings

1. The holy grail: Off-chain polling with on-chain execution (Aragon)
<https://aragon.org/blog/snapshot>
2. Decentralized Governance. How to Put Power Into the Hands of the People (Trust Wallet)
<https://trustwallet.com/blog/decentralized-governance-power-in-hands-of-people>
3. DAO or Die: How to Fully Decentralize the Off-chain Governance of Your Crypto Project (Otonomos)
<https://otonomos.com/2020/05/dao-or-die-how-to-fully-decentralize-the-off-chain-governance-of-your-crypto-project/>

CHAPTER 14: DEFI DASHBOARD

What is a Dashboard?

A dashboard is a simple platform that aggregates all your DeFi activities in one place. It is a useful tool to visualize and track where your assets are across the different DeFi protocols. The dashboard is able to segregate your assets into different categories such as deposit, debt and investments.

Typically when you access your dashboard, you will need to enter your Ethereum address (e.g.: `0x4Cd86fa95Ec2704f0849825f1F8b077deeD8d39`). Alternatively, you could enter your Ethereum Name Service (ENS) domain. ENS domain is a human-readable Ethereum address that you can purchase for a period of time. It is similar to Internet domain names such as <http://www.coingecko.com/> which then maps to the IP address of the server where CoinGecko is hosted.

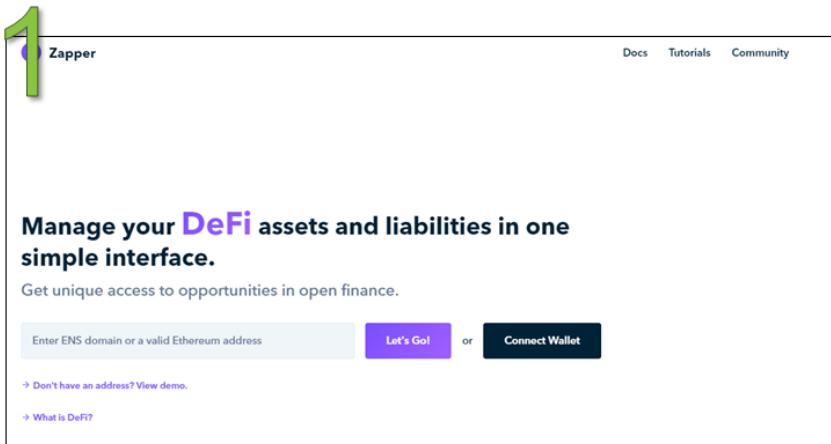
Check out our [ENS guide here](#) if you are interested in creating your own ENS domain!

Note: ENS domain is completely optional.

There are several dashboards in the market with the capacity to track your assets, but the two leading players for dashboards are Zapper (previously known as DeFiSnap) and Zerion.

For simplicity, we will be focusing on a step-by-step guide on how to use Zapper.

Zapper: Step-by-Step Guide



Step 1

- Go to <https://zapper.fi/>
- Enter your ENS domain or your Ethereum Address
- Here we used defiportal.eth but we can also key in 0x358a6c0f7614c44b344381b0699e2397b1483252

DeFi Dashboard

The screenshot shows the DeFi Zapper dashboard for the delportal.eth Ethereum network. At the top, there are three summary boxes: Total Assets (\$2,689.40), Total Debt (\$2.44), and Net Worth (\$2,686.96). Below this is the Account Overview section, which includes Wallet (\$1,365.73), Deposits (\$1,255.07), Investments (\$68.38), Yield Farming (\$0.23), and Debt (\$2.44). A 'Switch View' button is also present. The Platforms section lists Synthetix (\$9.82), Compound (\$7.43), Aave (\$10.86), dYdX (\$204.63), and PoolTogether (\$1.00). The Asset Allocation section shows the distribution of assets across different categories, while the Platform Allocation section shows the distribution of assets across various DeFi protocols.

Category	Value
Total Assets	\$2,689.40
Total Debt	\$2.44
Net Worth	\$2,686.96
Wallet	\$1,365.73
Deposits	\$1,255.07
Investments	\$68.38
Yield Farming	\$0.23
Debt	\$2.44
Synthetix	\$9.82
Compound	\$7.43
Aave	\$10.86
dYdX	\$204.63
PoolTogether	\$1.00

Category	Allocation (%)
Wallet	50.78%
Deposits	46.67%
Investments	2.54%
Yield Farming	0.01%
dYdX	87.46%
Aave	4.64%
Synthetix	4.20%
Compound	3.18%
PoolTogether	0.43%

Step 2

- You are on the dashboard!
- You can see your wallet balance, and any DeFi deposits, investments, yield farming, and debt.
- Scroll down further, and you will see which protocols your assets are currently locked inside.

How to DeFi: Beginner

The image contains three screenshots of the Zapper interface, each with a large green number 3 in the top-left corner.

- Top Screenshot (Dashboard):** Shows the main dashboard with a "Waller not connected" message and a "Connect Wallet" button. It includes tabs for Dashboard, Exchange, Pool, Farm, and History. A purple bar at the bottom has "Exchange" highlighted.
- Middle Screenshot (Farm):** Shows the "Farm" section with a red box highlighting the "Farm" tab. It displays a table of available pools, their liquidity, volume, fees, and "Add Liquidity" buttons. The table includes rows for WBTC / ETH, 3Pool Curve, renBTC Curve, and sBTC Curve.
- Bottom Screenshot (Farm):** Shows the "Farm" section with a red box highlighting the "Farm" tab. It displays a table of farming opportunities, showing assets, liquidity, ROI, and rewards. The table includes rows for WBTC / ETH, renBTC Curve, sBTC Curve, and SUSHI / ETH.

Step 3

- Apart from seeing your asset allocation on-chain, Zapper serves as an all-in-one dashboard for you to make your investment decisions, such as swapping tokens, providing liquidity, and participating in yield farms.

Alternatively, you can check out other dashboards:

- <https://zerion.io/>
- <https://mydefi.org/apps> (acquired by Zerion but still operating)
- <https://frontierwallet.com/> (Dashboard for mobile phones)
- <https://debank.com/>
- <https://unspent.io/>

PART FOUR: DEFI IN ACTION

CHAPTER 15: DEFI IN ACTION

In the previous sections, we talked about the importance of DeFi and showed some of the products available in the DeFi ecosystem. However, questions remain on just how decentralized DeFi Dapps are and if anyone is actually using DeFi in real life. In this section, we will explore DeFi in action with two case studies showing the robustness and usefulness of DeFi.

Surviving Argentina's High Inflation

During the Devcon 5 Ethereum conference in October 2019, Mariano Conti, Head of Smart Contracts at Maker Foundation gave a [talk](#) on how he survives Argentina's inflation. The inflation rate for Argentina reached 53.8% in 2019, the highest rate in 28 years. This placed Argentina as the top 5 countries in the world with the highest inflation rates.¹⁰

To live in a country where the value of your national currency practically halves every year is tough. To survive in Argentina, Mariano requested for his salary to be fully paid in DAI. As you may recall from our previous section, DAI is a stablecoin pegged to the USD. According to Mariano, Argentinians value the USD a lot. Despite the USD having inflationary problems as well, compared to the Argentine Peso, it is nothing.

¹⁰ "Argentina inflation expected at 53% in December ... – Reuters." 11 Sep. 2019, <https://www.reuters.com/article/argentina-economy/argentina-inflation-expected-at-53-in-december-2019-treasury-officials-idINKCN1VX09U>.

Argentina Inflation Rate (1995 - 2020)



Source: [TradingEconomics.com](https://tradingeconomics.com/argentina/inflation-rate)

If the USD is attractive to most Argentinians, it is natural then that most Argentinians would prefer to keep their money in USD. However, the government in Argentina places capital control on this, making it hard to get access to the USD. There is a limit on the purchase of USD, and Argentinians can only purchase a maximum of \$200 per month. As a result of this, the black market demand for the USD has risen, causing the exchange rate to be approximately 30% higher than the officially declared rate by the government.¹¹

Besides placing a limit on purchases, the Central Bank of Argentina also exposed 800 citizens' names, ID number and tax identification because they exceeded the previous purchase limit of \$10,000.¹² Furthermore, Argentinians who work for foreign companies and are invoiced in USD must liquidate their USD to Argentine Peso within 5 days.

According to Mariano, several years ago, many Argentinian freelancers preferred getting paid in Bitcoin. While this worked well in the earlier years

¹¹ “Argentina’s ‘little trees’ blossom as forex controls fuel black ...” 5 Feb. 2020, <https://www.reuters.com/article/us-argentina-currency-blackmarket/argentinas-little-trees-blossom-as-forex-controls-fuel-black-market-idUSKBN1ZZ1H1>.

¹² “Argentina Central Bank Exposed 800 Citizens ... – BeInCrypto.” 29 Sep. 2019, <https://beincrypto.com/argentina-central-bank-exposed-sensitive-information-of-800-citizens/>.

before 2018 when Bitcoin price was on an uptrend, as the market turned downwards, there was an urgent need to convert Bitcoin immediately to Argentine Peso otherwise their salary will be greatly reduced. While Bitcoin provided many Argentinians with an alternative way of being paid, the volatile nature of Bitcoin meant that there was a need for “better money”.

For Mariano, DAI is the solution to this problem as it has all the advantages of cryptocurrencies while staying pegged to the USD. But what does he do with his DAI? Once a month, he withdraws the bare minimum to pay for items like rent, groceries, and credit card bills, keeping his Argentine Peso balance as close to 0 as possible.

He also uses his DAI for crypto transactions, such as purchasing ETH and putting DAI into Dai Savings Rate. From this, he can earn interest on a stablecoin that he would otherwise not have access to. While he acknowledges that by using DeFi Dapps, he exposes himself to smart contract risks, he feels that the risk of holding Argentinian Peso is high too.

To Mariano, being paid in DAI allows him to “escape” issues such as volatility, inflation, and control facing his country. This issue is not just facing Argentina but several other economies globally, and this is proof of how DeFi can be valuable for people living in these countries.

Click the link below to watch Mariano’s entire presentation:

<https://slideslive.com/38920018/living-on-defi-how-i-survive-argentinas-50-inflation>.

Uniswap Ban

This content is not available in your country due to a legal complaint from the government. [Learn more.](#)

Sorry about that.



Looks familiar? (Image Credit: gtricks.com¹³)

Most of us have likely seen this—a video or mobile app that is not available because of our location or due to censorship. It's infuriating and annoying, but life goes on. You can find the video elsewhere or simply download another similar application for the same service.

Bans on videos or apps may not have overly adverse effects, but the same cannot be said if you are banned from having access to banks or financial institutions instead. Unfortunately, this hits hardest on those who need it most, as they likely do not keep large amounts of extra cash on hand. A person may be forced to take loans to cover their expenses, potentially snowballing to many other things down the road.

That being said, within the DeFi ecosystem, censorships have happened as well. One notable occasion was the Uniswap (decentralized exchange) geographical ban back in December 2019. At that time, the Uniswap team quietly updated and published a change to their open-sourced codebase on Github to exclude certain countries (Belarus, Cuba, Iran, Iraq, Côte d'Ivoire,

¹³ “Watch YouTube Blocked Videos Not Available in Your Country.” <https://www.gtricks.com/youtube/watch-blocked-youtube-videos-not-available-in-your-country/>. Accessed 27 Feb. 2020.

How to DeFi: Beginner

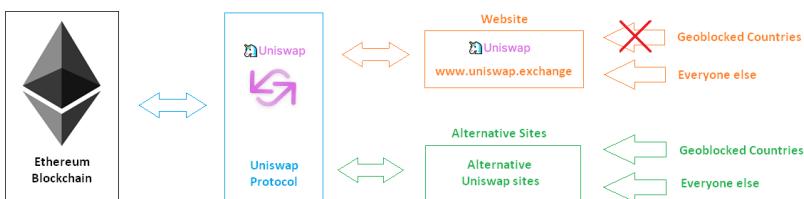
Liberia, North Korea, Sudan, Syria, Zimbabwe) from accessing the main website at www.uniswap.exchange.¹⁴ The result looks like this:



With the geoblock in place, people from the blocked countries can no longer access the Uniswap.exchange website.

Word on the street is that the Uniswap team did so to remain compliant with US laws as the team is based in New York. Regardless of the reason, if Uniswap were to make their services inaccessible to people because of their location, it would go against everything DeFi stands for - allowing **anyone** to have access from **anywhere**.

In upholding the true spirits of DeFi, the geographical ban that the Uniswap team imposed did not stop users from using the Uniswap protocol - in fact, they couldn't. The Uniswap protocol is built and deployed on the Ethereum blockchain, which is accessible worldwide by anyone. Within a matter of hours, multiple sites that are connected to the Uniswap protocol have gone live, enabling banned users to continue accessing the Uniswap protocol.



Since Uniswap protocol is permissionless, anyone can connect to it if they know how to, or have an interface (like uniswap.exchange website) that allows them to.

The key point to note in this incident is that, while the Uniswap team had control of the front-end (www.uniswap.exchange), they had no control over

¹⁴ “Uniswap/uniswap-frontend: An open-source ... - GitHub.”

<https://github.com/Uniswap/uniswap-frontend>. Accessed 27 Feb. 2020.

who can or cannot access the back-end (Uniswap protocol) deployed onto the Ethereum blockchain.

This was an interesting case study as it highlighted the strengths of DeFi protocols, something that would not happen in traditional finance. **A move that was initially made to go against the very core ethos of DeFi ended up showcasing one of its key strengths.**

This is not the first and will not be the last time DeFi applications are challenged. It will be exciting to see what the future holds!

CHAPTER 16: DEFI IS THE FUTURE, AND THE FUTURE IS NOW

The previous chapter highlighted two examples of DeFi in action when DeFi was still at its nascent stage. As DeFi continues to grow, we are already seeing a massive influx of institutional and retail investors. Many pundits from traditional finance have already started to recognize bitcoin's value, which is often equated to digital gold. However, if you have read this book, you would understand that bitcoin is merely scratching the surface of what DeFi truly offers.

We are already starting to see Traditional Finance (TradFi) players tap into the DeFi market. For example, [Siam Commercial Bank](#) has invested in Alpha Finance Lab and is working closely together to create a unique suite of products that bridges the gap between the traditional and decentralized financial sectors. Other examples include [Yield.app](#) that doubles up as a yield aggregator (similar to hedge funds) and employs both portfolio managers and smart contract technology for users to earn passive income in DeFi. As bridges between TradFi and DeFi start to develop, DeFi is increasingly recognized as an alternative financial ecosystem.

The overarching theme is that TradFi cannot ignore the lucrative opportunities that DeFi offers because DeFi represents the future of finance. While this can sound controversial, we will summarize here why we think this is the case.

At the start of 2020, the Total Value Locked in DeFi Dapps hit the significant milestone of \$1 billion. In other words, that is the total amount of programmable money currently stored in smart contracts that serve as the building blocks of an entirely new decentralized financial system on the internet.

While \$1 billion locked into DeFi may seem like a small number compared to traditional financial markets, the growth has been staggering. Here's a quick summary of the journey:

- 2018: Total Value Locked increased 5 times from \$50 million to \$275 million
- 2019: Total Value Locked increased 2.4 times to \$667 million
- 2020: Total Value Locked increased 23.5 times to \$15.7 billion
- 2021: Total Value Locked reached \$86.05 billion (April 2021)¹⁵

Before we push on, let's have a quick recap on some of the things DeFi allows us to achieve:

Transparency: A transparent, auditable financial ecosystem.

Accessibility: Free access to DeFi applications without fear of discrimination on race, gender, beliefs, nationality, or geographical status.

Efficiency: Programmable money makes it possible to remove the centralized middlemen to create a more affordable and efficient financial market.

Convenience: Money can now be sent anywhere, anytime, and to anyone with a cryptocurrency wallet. All this for a small fee and with little waiting time.

All of the above have made it possible for users to do provide liquidity to earn yields on unproductive assets with no maturation/lock-in period, take loans (with collateral) without paperwork and repay them anytime, and execute automated trading strategies easily.

¹⁵ <https://defillama.com/home>

DeFi allows users to access financial services anywhere and anytime, as long as one is connected to the Internet. Now that's the power of DeFi accessibility, and we are only just getting started on this journey.

What about DeFi User Experience?

We are glad you asked—while accessibility to DeFi apps may be a non-issue, one of the major pain points for DeFi remains the overall user experience.

That said, many teams worldwide are hard at work trying to improve the experience. Have a look at some of them and the aspects they are attempting to solve:

Wallet - [Argent](#) is creating a radically better user-focused crypto wallet experience, with state-of-the-art security, native integration with DeFi Dapps such as Compound and others, as well as not needing seed keys.

Participation of products - [Zapper](#) abstracts away many of the complexities and steps involved with DeFi products. It also allows users to access multiple financial products in one transaction, saving time and effort.

User-friendly development - [Gelato Finance](#) recently launched their “If this, then that” for crypto. It allows users to set actions that will be done once certain conditions are met, such as “Buy ETH when it is \$200” or “Send some money to Alice when it’s her birthday.”

Insurance - Financial market effectively facilitates the transfer of risk - one man’s hedge against his position is another man’s profit. Insurance is now available via DeFi insurers such as [Nexus Mutual](#). If you are willing to accept a lower yield on the money you have placed on lending protocols such as Compound in exchange for peace of mind, it can now be done.

Aggregation of liquidity - There are many different decentralized exchanges (DEXs) in the market with varying liquidity. It is a headache for users to choose which one is the best for their trade. This is slowly becoming a thing of the past with liquidity aggregators such as [1inch.exchange](#),

[Paraswap](#), and [Matcha](#) helping users to automatically split orders across DEXs to ensure the best possible execution prices.

Yield optimization - Remember switching around different banks for the best rates for fixed deposits? You don't have to do that in DeFi - yield aggregators such as [Yearn.finance](#), [idle.finance](#), and [DeFiSaver](#) automatically allocate your cryptoassets to places with the best yield opportunities.

While there is no single “killer app” that bridges the user experience gap at the moment, we think it's not going to be far away!

CLOSING REMARKS

Phew, that was a blast to write! If you are reading this line here, congratulations, you are now up to date on DeFi, and you should pat yourself on the back!

Thank you for your time, and we hope you have enjoyed reading this book as much as we have enjoyed researching, learning, and writing it! :)

Welcome to DeFi and the future of finance!

APPENDIX

CoinGecko's Recommended DeFi Resources

Information

DefiLlama - <https://defillama.com/home>

DeBank - <https://debank.com/>

DeFi Prime - <https://defiprime.com/>

DeFi Pulse - <https://defipulse.com/>

LoanScan - <http://loanscan.io/>

News Sites

CoinDesk - <https://www.coindesk.com/>

CoinTelegraph - <https://cointelegraph.com/>

Decrypt - <https://decrypt.co/>

The Block - <https://www.theblockcrypto.com/>

Crypto Briefing - <https://cryptobriefing.com/>

Newsletters

Bankless - <https://bankless.substack.com/>

DeFi Tutorials - <https://defitutorials.substack.com/>

DeFi Weekly - <https://defiweekly.substack.com/>

Dose of DeFi - <https://doseofdefi.substack.com/>

Ethhub - <https://ethhub.substack.com/>

My Two Gwei - <https://mytwogwei.substack.com/>

The Defiant - <https://thedefiant.substack.com/>

Week in Ethereum News - <https://www.weekinethereumnews.com/>

Podcast

CoinGecko - <https://podcast.coingecko.com/>

BlockCrunch - <https://castbox.fm/channel/Blockcrunch%3A-Crypto-Deep-Dives-id1182347>

Chain Reaction - <https://fiftyonepercent.podbean.com/>

Into the Ether - Ethhub - <https://podcast.ethhub.io/>

PoV Crypto - <https://povcryptopod.libsyn.com/>

Wyre Podcast - <https://blog.sendwyre.com/wyretalks/home>

Youtube

Yield TV by Zapper -

<https://www.youtube.com/channel/UCYq3ZxBx7P2ckJyWVDC597g>

Bankless -

<https://www.youtube.com/channel/UCAI9Ld79qaZxp9JzEOwd3aA>

Chris Blec - <https://www.youtube.com/c/chrisblec>

Bankless Level-Up Guide

<https://bankless.substack.com/p/bankless-level-up-guide>

Projects We Like Too

Dashboard Interfaces

Zapper - <https://zapper.fi/dashboard>

Frontier - <https://frontierwallet.com/>

InstaDapp - <https://instadapp.io/>

Zerion - <https://zerion.io/>

Debank - <https://debank.com/>

Decentralized Exchanges

SushiSwap - <https://sushi.com/>

Balancer - <https://balancer.exchange/>

Bancor - <https://www.bancor.network/>

Curve Finance - <https://www.curve.fi/>

Kyber Network - <https://kyberswap.com/swap>

Appendix

Exchange Aggregators

1inch - <https://1inch.exchange/>

Dex.ag - <https://dex.ag/>

Paraswap - <https://paraswap.io/>

Matcha - <https://matcha.xyz/>

Lending and Borrowing

Compound - <https://compound.finance/>

Aave - <https://aave.com/>

Cream - <https://cream.finance/>

Prediction Markets

Augur - <https://www.augur.net/>

Taxes

TokenTax - <https://tokentax.co/>

Wallet

GnosisSafe - <https://safe.gnosis.io/>

Monolith - <https://monolith.xyz/>

Yield Optimisers

Yearn - <https://yearn.finance/>

Alpha Finance - <https://alphafinance.io/>

References

Chapter 1: Traditional Financial Institutions

Bagnall, E. (2019, June 30). Top 1000 World Banks 2019 – The Banker International Press Release – for immediate release. Retrieved February 20, 2020, from <https://www.thebanker.com/Top-1000-World-Banks/Top-1000-World-Banks-2019-The-Banker-International-Press-Release-for-immediate-release>

Boehlke, J. (2019, September 18). How Long Does It Take to Have a Payment Post Online to Your Bank? Retrieved February 20, 2020, from <https://www.gobankingrates.com/banking/checking-account/how-long-payment-posted-online-account/>

Demirguc-Kunt, A., Klapper, L., Singer, D., Ansar, S., Hess, J. (2018). The Global Findex Database 2017: Measuring Financial Inclusion and the Fintech Revolution. https://doi.org/10.1596/978-1-4648-1259-0_ch2

How long does an Ethereum transaction really take? (2019, September 25). Retrieved February 20, 2020, from <https://ethgasstation.info/blog/ethereum-transaction-how-long/>

International Wire Transfers. (n.d.). Retrieved February 20, 2020, from <https://www.bankofamerica.com/foreign-exchange/wire-transfer.go>

Karlan, D., Ratan, A. L., & Zinman, J. (2014, March). Savings by and for the poor: a research review and agenda. Retrieved February 20, 2020, from <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4358152/>

Stably. (2019, September 20). Decentralized Finance vs. Traditional Finance: What You Need To Know. Retrieved from <https://medium.com/stably-blog/decentralized-finance-vs-traditional-finance-what-you-need-to-know-3b57aed7a0c2>

Appendix

Chapter 2: What is Decentralized Finance (DeFi)?

Campbell, L. (2020, January 6). DeFi Market Report for 2019 - Summary of DeFi Growth in 2019. Retrieved from <https://defirate.com/market-report-2019/>

Mitra, R. (n.d.). DeFi Use cases: The Best Examples of Decentralised Finance. Retrieved from https://blockgeeks.com/guides/defi-use-cases-the-best-examples-of-decentralised-finance/#_Tool_2_DeFi_Derivatives

Shawdagor, J. (2020, February 23). Sectors Realizing the Full Potential of DeFi Protocols In 2020. Retrieved from
<https://cointelegraph.com/news/sectors-realizing-the-full-potential-of-defi-protocols-in-2020>

Thompson, P. (2020, January 5). Most Significant Hacks of 2019 - New Record of Twelve in One Year. Retrieved February 20, 2020, from
<https://cointelegraph.com/news/most-significant-hacks-of-2019-new-record-of-twelve-in-one-year>

Chapter 3: The Decentralized Layer: Ethereum

What is Ethereum? (2020, February 11). Retrieved from
<https://ethereum.org/what-is-ethereum/>

Rosic, A. (2018). What is Ethereum Gas? [The Most Comprehensive Step-By-Step Guide]. Retrieved from
<https://blockgeeks.com/guides/ethereum-gas/>

Rosic, A. (2017). What Are Smart Contracts? [Ultimate Beginner's Guide to Smart Contracts]. Retrieved from <https://blockgeeks.com/guides/smart-contracts/>

Chapter 4: Ethereum Wallets

Lee, I. (2018, June 22). A Complete Beginner's Guide to Using MetaMask. Retrieved from <https://www.coingecko.com/buzz/complete-beginners-guide-to-metamask>

Lesuisse, I. (2018, December 22). A new era for crypto security. Retrieved from <https://medium.com/argenthq/a-new-era-for-crypto-security-57909a095ae3>

Wright, M. (2020, February 13). Argent: The quick start guide. Retrieved from <https://medium.com/argenthq/argent-the-quick-start-guide-13541ce2b1fb>

Chapter 5: Decentralized Stablecoins

The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System (n.d.). Retrieved February 20, 2020, from <https://makerdao.com/whitepaper/>

MKR Tools (n.d.). Retrieved February 20, 2020, from <https://mkr.tools/governance/stabilityfee>

Maker Governance Dashboard (n.d.). Retrieved February 20, 2020, from <https://vote.makerdao.com/pollin>

Currency Re-imagined for the World: Multi-Collateral Dai Is Live! (2019, November 18). Retrieved from <https://blog.makerdao.com/multi-collateral-dai-is-live/>

Dai is now live! (2017, December 19). Retrieved from <https://blog.makerdao.com/dai-is-now-live/>

DSR. (n.d.). Retrieved February 20, 2020, from <https://community-development.makerdao.com/makerdao-mcd-faqs/faqs/dsr>

John, J. (2019, December 4). Stable Coins In 2019. Retrieved from <https://www.decentralised.co/what-is-going-on-with-stable-coins/>

Tether: Fiat currencies on the Bitcoin blockchain. (n.d.). Tether Whitepaper. Retrieved from <https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf>

Appendix

Chapter 6: Decentralized Borrowing and Lending

Kulechov, S. (2020). The Aave Protocol V2. Retrieved 28 January 2021, from <https://medium.com/aave/the-aave-protocol-v2-f06f299cee04>

Leshner, R. (2018, December 6). Compound FAQ. Retrieved from <https://medium.com/compound-finance/faq-1a2636713b69>

(2021). Retrieved 28 January 2021, from <https://docs.aave.com/portal/>
(2021). Retrieved 28 January 2021, from
<https://github.com/aave/governance-v2>

Chapter 7: Decentralized Exchange (DEX)

Connect to Uniswap. (n.d.). Retrieved from
<https://docs.uniswap.io/frontend-integration/connect-to-uniswap#factory-contract>

Introducing 1inch v2. Retrieved 28 January 2021, from
<https://1inch-exchange.medium.com/introducing-1inch-v2-defis-fastest-and-most-advanced-aggregation-protocol-c42573dc3f85>

Peaster, W. (2020). Initial DeFi Offering. Retrieved from
<https://defiprime.com/initial-defi-offering>

Uniswap: Stats, Charts and Guide: DeFi Pulse. (n.d.). Retrieved from
<https://defipulse.com/uniswap>

Uniswap Whitepaper. (n.d.). Retrieved from
<https://hackmd.io/@Uniswap/Hj9jLsfTz>

Zhang, Y., Chen, X., & Park, D. (2018). Formal Specification of Constant Product ($x \times y = k$) Market Maker Model and Implementation. Retrieved from <https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf>

Chapter 8: Decentralized Derivatives

Tulip Mania (n.d.). Retrieved from

https://penelope.uchicago.edu/~grout/encyclopaedia_romana/aconite/tulipomania.html

Chen, J. (2020, January 27). Derivative. Retrieved from

<https://www.investopedia.com/terms/d/derivative.asp>

Decentralised synthetic assets. (n.d.). Retrieved from

<https://www.synthetix.io/products/exchange/>

Synthetix.Exchange Overview. (2019, February 15). Retrieved from

<https://blog.synthetix.io/synthetix-exchange-overview/>

Synthethix Litepaper v1.3. (2019). Retrieved from

https://www.synthetix.io/uploads/synthetix_litepaper.pdf

Chapter 9: Decentralized Fund Management

Making Sense of the Mutual Fund Scandal Everything you may not want to ask (but really should know) about the crisis that's rocking the investment world. (2003, November 24). Retrieved from

https://money.cnn.com/magazines/fortune/fortune_archive/2003/11/24/353794/index.htm

The Editors of Encyclopaedia Britannica. (2020, February 26). Bernie Madoff. Retrieved from <https://www.britannica.com/biography/Bernie-Madoff>

Frequently Asked Questions on TokenSets. (n.d.). Retrieved from

<https://www.tokensets.com/faq>

Liang, R. (2019, April 23). TokenSets is Live: Automate your Crypto Portfolio Now. Retrieved from <https://medium.com/set-protocol/tokensets-is-live-automate-your-crypto-portfolio-now-50f88dcc928d>

Appendix

Sawinyh, N. (2019, June 17). Interview with TokenSets creators. Retrieved from <https://defiprime.com/tokensets>

Sassano, A. (2019, June 19). How Set Protocol Works Under the Hood. Retrieved from <https://medium.com/@AnthonySassano/how-set-protocol-works-under-the-hood-74fcdae858e2>

Sassano, A. (2020, January 22). Set Social Trading is Now Live on TokenSets. Retrieved from <https://medium.com/set-protocol/set-social-trading-is-now-live-on-tokensets-c981b5e67c5f>

Sassano, A. (2020). What To Expect With Set V2. Retrieved 28 January 2021, from <https://medium.com/set-protocol/what-to-expect-with-set-v2-15459581c6d4>

Chapter 10: Decentralized Lottery

Cusack, L. (2020, February 3). PoolTogether raises \$1 Million to Expand Prize Linked Savings Protocol. Retrieved from <https://medium.com/pooltogether/pooltogether-raises-1-million-to-expand-prize-linked-savings-protocol-eb51a1f88ed8>

Guillén, M.F., Tschoegl, A.E. Banking on Gambling: Banks and Lottery-Linked Deposit Accounts. Journal of Financial Services Research 21, 219–231 (2002). <https://doi.org/10.1023/A:1015081427038>

H.148. (2019). Retrieved from <https://legislature.vermont.gov/bill/status/2020/H.148>

Lemke, T. (2019, February 21). What Are Prize-Linked Savings Accounts? Retrieved from <https://www.thebalance.com/what-are-prize-linked-savings-accounts-4587608>

LLC, P. T. (n.d.). PoolTogether. Retrieved from <https://www.pooltogether.com/#stats>

Markets. (n.d.). Retrieved from <https://compound.finance/markets>

PoolTogether. (2020, February 8). Wow! The winner of the largest prize ever only 10 Dai deposited! They won \$1,648 Dai A 1 in 69,738 chance of winning. Congrats to the little fish! pic.twitter.com/0DSFkSdbIE. Retrieved from

[https://twitter.com/PoolTogether /status/1225875154019979265](https://twitter.com/PoolTogether/status/1225875154019979265)

Texas Proposition 7, Financial Institutions to Offer Prizes to Promote Savings Amendment (2017). (2017). Retrieved from [Texas Proposition 7, Financial Institutions to Offer Prizes to Promote Savings Amendment \(2017\)](#)

Chapter 11: Decentralized Payment

Bramanathan, R. (2020, February 1). What I learned from tokenizing myself. Retrieved from <https://medium.com/@bramanathan/what-i-learned-from-tokenizing-myself-bb222da07906>

Chapter 12: Decentralized Insurance

Blockchain, F. (2019, December 4). The Potential for Bonding Curves and Nexus Mutual. Retrieved from
<https://tokentuesdays.substack.com/p/the-potential-for-bonding-curves>

Blockchain, F. (2019, October 2). Nexus Mutual. Retrieved from
<https://tokentuesdays.substack.com/p/nexus-mutual>

Codefi Data. (n.d.). Retrieved from <https://defiscore.io/>

defidad.eth, D. F. D.-. (2020, February 11). @NexusMutual is a decentralized alternative to insurance, providing the #Ethereum community protection against hacks. Here's why it should be on your radar: + Anyone can buy smart contract insurance + Being a backer (staker) can earn up to 50% ROI + It's powered by #Ethereum. Retrieved from

https://twitter.com/DeFi_Dad/status/1227165545608335360?s=09

Docs. (n.d.). Retrieved from

<https://nexusmutual.gitbook.io/docs/docs#pricing>

Appendix

Karp, H. (2019, May 22). Nexus Mutual Audit Report. Retrieved from <https://medium.com/nexus-mutual/nexus-mutual-audit-report-57f1438d653b>

Karp, H. (2019, June 5). Nexus Mutual NXM Token Explainer. Retrieved from <https://medium.com/nexus-mutual/nexus-mutual-nxm-token-explainer-b468bc537543>

Russo, C. (2020, February 19). Arbs made ~\$900K in seconds by exploiting DeFi. It's mind-blowing stuff. Here's The Defiant post w/ exploits' twisted steps (in pics), qs raised about decentralization and price oracles, and consequences so far. What's your take on the blame game? Retrieved from <https://twitter.com/CamiRusso/status/1229849049471373312>

Token Model. (n.d.). Nexus Mutual: A decentralised alternative to insurance. Retrieved from <https://nexusmutual.io/token-model>

Welcome to the Nexus Mutual Gitbook. (n.d.). Retrieved from <https://nexusmutual.gitbook.io/docs/>

Coingecko. (2019). CoinGecko Quarterly Report for Q3 2019. Retrieved from <https://assets.coingecko.com/reports/2019-Q3-Report/CoinGecko-2019-Q3-Report.pdf>

Defiprime. (2020, February 13). what's the key difference vs. @NexusMutual ? Retrieved from <https://twitter.com/defiprime/status/1227720835898560513>

Karp, H. (2019, November 15). Comparing Insurance Like Solutions in DeFi. Retrieved from https://medium.com/@hugh_karp/comparing-insurance-like-solutions-in-defi-a804a6be6d48

OpenZeppelin Security. (2020, February 10). Opyn Contracts Audit. Retrieved from <https://blog.openzeppelin.com/opyn-contracts-audit/>

Chapter 13: Governance

(2020, March 6). Aragon (ANT) Economics. Retrieved from
<https://www.placeholder.vc/blog/2020/3/6/aragon-ant-economics>

(2020, October 20). Proposal: 3 Ideas to Improve Court Security. Retrieved from <https://forum.aragon.org/t/proposal-3-ideas-to-improve-court-security/2377>

(n.d.). Welcome to Snapshot! Retrieved from <https://docs.snapshot.page/>

Chapter 14: DeFi Dashboard

Dashboard for DeFi. (n.d.). Retrieved from
<https://www.defisnap.io/#/dashboard>

Chapter 15: DeFi in Action

(n.d.). Retrieved October 19, 2019, from
<https://slideslive.com/38920018/living-on-defi-how-i-survive-argentinas-50-inflation>

Gundiuc, C. (2019, September 29). Argentina Central Bank Exposed 800 Citizens' Sensitive Information. Retrieved from
<https://beincrypto.com/argentina-central-bank-exposed-sensitive-information-of-800-citizens/>

Lopez, J. M. S. (2020, February 5). Argentina's 'little trees' blossom as forex controls fuel black market. Retrieved from
<https://www.reuters.com/article/us-argentina-currency-blackmarket/argentinas-little-trees-blossom-as-forex-controls-fuel-black-market-idUSKBN1ZZ1H1>

Russo, C. (2019, December 9). Uniswap Website Geo-Ban Can't Stop DeFi. Retrieved from <https://thedefiant.substack.com/p/uniswap-website-geo-ban-cant-stop-370>

GLOSSARY

Index	Term	Description
#		
A	Annual Percentage Yield (APY)	It is an annualized return on saving or investment and the interest is compounded based on the period.
	Admin Key Risk	It refers to the risk where the master private key for the protocol could be compromised.
	Automated Market Maker (AMM)	Automated Market Maker removes the need for a human to manually quote bid and ask prices in an order book and replaces it with an algorithm.
	Audit	Auditing is a systematic process of examining an organization's records to ensure fair and accurate information the organization claims to represent. Smart contract audit refers to the practice of reviewing the smart contract code to find vulnerabilities so that they can be fixed before it is exploited by hackers.
	An Application Programming Interface (API)	An interface that acts as a bridge that allows two applications to interact with each other. For example, you can use CoinGecko's API to fetch the current market price of cryptocurrencies on your website.

Index	Term	Description
B	Buy and Hold	This refers to a TokenSets trading strategy which realigns to its target allocation to prevent overexposure to one coin and spreads risk over multiple tokens.
	Bonding Curve	A bonding curve is a mathematical curve that defines a dynamic relationship between price and token supply. Bonding curves act as an automated market maker where as the number of supply of a token decreases, the price of the token increases. It is useful as it helps buyers and sellers to access an instant market without the need of intermediaries.
C	Cryptocurrency Exchange	It is a digital exchange that helps users exchange cryptocurrencies. For some exchanges, they also facilitate users to trade fiat currencies to cryptocurrencies.
	Custodian	Custodian refers to the third party to have control over your assets.
	Fiat-collateralized stablecoin	A stablecoin that is backed by fiat-currency. For example, 1 Tether is pegged to \$1.
	Crypto-collateralized stablecoin.	A stablecoin that is backed by another cryptocurrency. For example, Dai is backed by Ether at an agreed collateral ratio.
	Centralized Exchange (CEX)	Centralized Exchange (CEX) is an exchange that operates in a centralized manner and requires full custody of users' funds.
	Collateral	Collateral is an asset you will have to lock-in with the lender in order to borrow another asset. It acts as a guarantor that you will repay your loan.
	Collateral Ratio	Collateral ratio refers to the maximum amount of asset that you can borrow after putting collateral into a DeFi decentralized application.
	cTokens	cTokens are proof of certificates that you have supplied tokens to Compound's liquidity pool.

Glossary

Index	Term	Description
	Cryptoasset	Cryptoasset refers to digital assets on blockchain. Cryptoassets and cryptocurrencies generally refer to the same thing.
	Cover Amount	It refers to the maximum payable money by the insurance company when a claim is made.
	Claim Assessment process	It is the obligation by the insurer to review the claim filed by an insurer. After the process, the insurance company will reimburse the money back to the insured based on the Cover Amount.
	Composability	Composability is a system design principle that enables applications to be created from component parts.
D	Decentralized Finance (DeFi)	DeFi is an ecosystem that allows for the utilization of financial services such as borrowing, lending, trading, getting access to insurance, and more without the need to rely on a centralized entity.
	Decentralized Applications (Dapps)	Applications that run on decentralized peer-to-peer networks such as Ethereum.
	Decentralized Autonomous Organization (DAO)	Decentralized Autonomous Organizations are rules encoded by smart contracts on the blockchain. The rules and dealings of the DAO are transparent and the DAO is controlled by token holders.
	Decentralized Exchange (DEX)	Decentralized Exchange (DEX) allows for trading and direct swapping of tokens without the need to use a centralized exchange.
	Derivatives	Derivative comes from the word derive because it is a contract that derives its value from an underlying entity/product. Some of the underlying assets can be commodities, currencies, bonds, or cryptocurrencies.

Index	Term	Description
	Dai Saving Rate (DSR)	The Dai Savings Rate (DSR) is an interest earned by holding Dai over time. It also acts as a monetary tool to influence the demand of Dai.
	Dashboard	A dashboard is a simple platform that aggregates all your DeFi activities in one place. It is a useful tool to visualize and track where your assets are across the different DeFi protocols.
E	Ethereum	Ethereum is an open-source, programmable, decentralized platform built on blockchain technology. Compared to Bitcoin, Ethereum allows for scripting languages which has allowed for application development.
	Ether	Ether is the cryptocurrency that powers the Ethereum blockchain. It is the fuel for the apps on the decentralized Ethereum network
	ERC-20	ERC is an abbreviation for Ethereum Request for Comment and 20 is the proposal identifier. It is an official protocol for proposing improvements to the Ethereum network. ERC-20 refers to the commonly adopted standard used to create tokens on Ethereum.
	Exposure	Exposure refers to how much you are ‘exposed’ to the potential risk of losing your investment. For example, price exposure refers to the potential risk you will face in losing your investment when the price moves.
F	Future Contract	It is a contract which you enter to buy or sell a particular asset at a certain price at a certain date in the future.
	Factory Contract	It is a smart contract that is able to produce other new smart contracts.
G	Gas	Gas refers to the unit of measure on the amount of computational effort required to

Glossary

Index	Term	Description
H		execute a smart contract operation on Ethereum.
I	IDO	IDO stands for Initial Decentralized Exchange Offering or Initial DEX offering. This is where tokens are first offered for sale to the public using a DEXs liquidity pools.
	IMAP	IMAP stands for Internet Message Access Protocol. It is an Internet protocol that allows email applications to access email on TCP/IP servers.
	Impermanent Loss	Temporary loss of funds due to volatility leading to divergence in price between token pairs provided by liquidity providers.
Index		An index measures the performance of a basket of underlying assets. An index moves when the overall performance of the underlying assets in the basket moves.
Inverse		This Synthetix strategy is meant for those who wish to “short” a benchmark. Traders can purchase this when they think a benchmark is due to decrease.
J		-
K	Know-Your-Customer (KYC)	Know-Your-Customer (KYC) is a compliance process for business entities to verify and assess their clients.
L	Liquidation penalty	It is a fee that a borrower has to pay along with their liquidated collateral when the value of their collateral asset falls below the minimum collateral value.
	Liquidity Pools	Liquidity pools are token reserves that sit on smart contracts and are available for users to exchange tokens. Currently the pools are mainly used for swapping, borrowing, lending, and insurance.
	Liquidity Risk	A risk when protocols like Compound could run out of liquidity.

Index	Term	Description
	Liquidity Providers	Liquidity providers are people who loan their assets into the liquidity pool. The liquidity pool will increase as there are more tokens.
	Liquidity Pool Aggregator	It is a system which aggregates liquidity pools from different exchanges and is able to see all available exchange rates in one place. It allows you to compare for the best possible rate.
	Leverage	It is an investment strategy to gain higher potential return of the investment by using borrowed money.
M	MakerDAO	MakerDAO is the creator of Maker Platform and DAO stands for Decentralized Autonomous Organisation. MakerDAO's native token is MKR and it is the protocol behind the stablecoins, SAI and DAI.
	Market Maker Mechanisms	A Market Maker Mechanism is an algorithm that uses a bonding curve to quote both a buy and a sell price. In the crypto space, Market Maker Mechanism is mainly used by Uniswap or Kyber to swap tokens.
	Margin Trading	It is a way of investing by borrowing money from a broker to trade. In DeFi, the borrowing requires you to collateralize assets.
	MKR	Maker's governance token. Users can use it to vote for improvement proposal(s) on the Maker Decentralized Autonomous Organization (DAO).
	Mint	It refers to the process of issuing new coins/tokens.
N	-	-
O	Order book	It refers to the list of buying and selling orders for a specific asset at various price levels.

Glossary

Index	Term	Description
	Over-collateralization	Over-collateralization refers to the value of collateral asset that must be higher than the value of the borrowed asset.
	Option	Option is a right but not the obligation for someone to buy or sell a particular asset at an agreed price on or before an expiry date.
P	Price discovery	Price discovery refers to the act of determining the proper price of an asset through several factors such as market demand and supply.
	Protocol	A protocol is a base layer of codes that tells something on how to function. For example, Bitcoin and Ethereum blockchains have different protocols.
	Peer-to-Peer	In blockchain, “peer” refers to a computer system or nodes on a decentralized network. Peer-to-Peer (P2P) is a network where each node has an equal permission to validating data and it allows two individuals to interact directly with each other.
Q		-
R	Range Bound	This TokenSets strategy automates buying and selling within a designated range and is only intended for bearish or neutral markets.
	Rebalance	It is a process of maintaining a desired asset allocation of a portfolio by buying and selling assets in the portfolio.
	Risk Assessor	Someone who stakes value against smart contracts in Nexus Mutual. He/she is incentivized to do so to earn rewards in NXM token, as other users buy insurance on the staked smart contracts.
S	Smart Contracts	A smart contract is a programmable contract that allows two counterparties to set conditions of a transaction without needing to trust another third party for the execution.

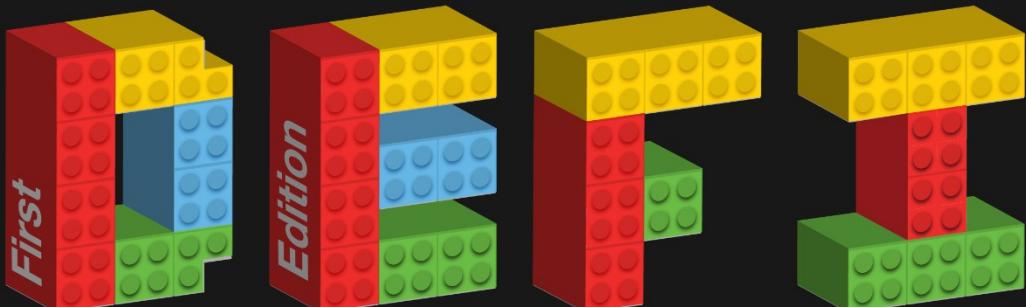
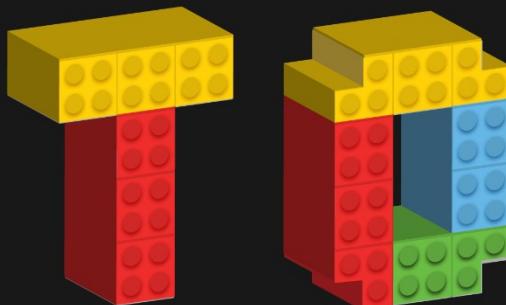
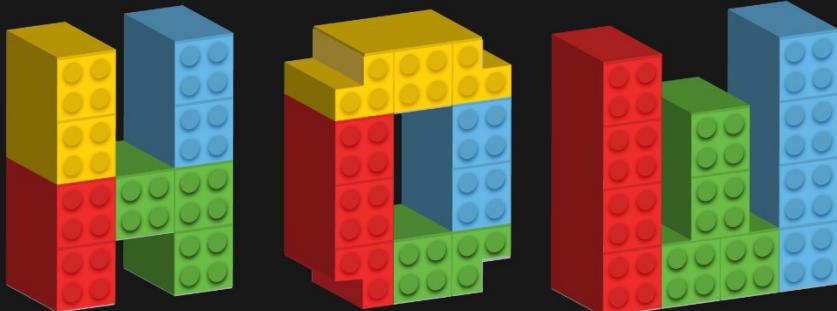
Index	Term	Description
	Stablecoins	A stablecoin is a cryptocurrency that is pegged to another stableasset such as the US Dollar.
	Spot market	Spot market is the buying and selling of assets with immediate delivery.
	Speculative activity	It is an act of buying and selling, while holding an expectation to gain profit.
	Stability Fee	It is equivalent to the ‘interest rate’ which you are required to pay along with the principal debt of the vault.
	Slippage	Slippage is the difference between the expected price and the actual price where an order was filled. It is generally caused by low liquidity.
	Synths	Synths stand for Synthetic Assets. A Synth is an asset or mixture of assets that has/have the same value or effect as another asset.
	Smart Contract Cover	An insurance offer from Nexus Mutual to protect users against hacks in smart contracts that stores value.
T	TCP/IP	It stands for Transmission Control Protocol/Internet Protocol. It is a communication protocol to interconnect network devices on the internet.
	Total Value Locked	Total Value Locked refers to the cumulative collateral of all DeFi products.
	Technical Risk	It refers to the bugs on smart contracts which can be exploited by hackers and cause unintended consequences.
	Trading Pairs	A trading pair is a base asset that is paired with its target asset in the trading market. For example, for the ETH/DAI trading pair, the base asset is ETH and its target pair is DAI.
	Trend Trading	This strategy uses Technical Analysis indicators to shift from 100% target asset to

Glossary

Index	Term	Description
		100% stableasset based on the implemented strategy.
	Tokens	It is a unit of a digital asset. Token often refers to coins that are issued on existing blockchain.
	Tokenize	It refers to the process of converting things into digital tradable assets.
U		-
V	Value Staked	It refers to how much value the insurer will put up against the target risk. If the value that the insurer staked is lower than the target risk, then it is not coverable.
W	Wallet	A wallet is a user-friendly interface to the blockchain network that can be used as a storage, transaction and interaction bridge between the user and the blockchain.
X		-
Y		-
Z		-

"Probably the most comprehensive DeFi manual out there,
a must-read."

Hugh Karp, Founder of Nexus Mutual



ADVANCED

Decentralized Finance is taking over the world.
Learn how to get started and join the revolution.



CoinGecko

How to DeFi: Advanced

1st Edition, May 2021

Lucius Fang, Benjamin Hor, Erina Azmi,
Khor Win Win

Copyright © 2021 CoinGecko
1st edition, May 2021

Layout: Anna Tan
teaspoonpublishing.com.my

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except brief extracts for the purpose of review, without the prior permission in writing of the publisher and copyright owners. It is also advisable to consult the publisher if in any doubt as to the legality of any copying which is to be undertaken.

“Probably the most comprehensive DeFi manual out there, a must-read.”

– Hugh Karp, Founder of Nexus Mutual

“Education is paramount in DeFi and resources such as How to DeFi are so important. Not only is this an excellent sequel, but once again, the team at CoinGecko have managed to provide a comprehensive and in-depth overview of an ever-changing space.”

– Ganesh Swami, CEO of Covalent

“This is the most comprehensive guide on DeFi anywhere, bar none. You should read this cover to cover.”

– Leo Cheng, Co-founder of CREAM Finance

“If you want to learn the latest trends in DeFi, this book is the best of the best on the market.”

– Yenwen Feng, Co-founder of Perpetual Protocol

“This book comes as an excellent follow-up to their first book, and provides a deeper dive into DeFi and on how to navigate the nuances in the space.”

– Jocelyn Chang, APAC Growth Lead of MakerDAO Growth Core Unit

“Reading *How To DeFi* in 2021 is like accidentally meeting Vitalik Buterin by chance in a café in Zug in 2015 and discovering Ethereum first-hand. *How To DeFi* will help you make life-changing decisions when building and using DeFi protocols and applications of this decade.”

– Molly Wintermute, Founder of Hegic

“DeFi isn’t easy for newcomers—yet. But with this top CoinGecko guide, readers will very quickly discover how DeFi is not just the future. DeFi is now, and soon it will become a regular part of everyday life for many people across the world. This is probably the best guide right now to light the way on anyone’s DeFi journey at any stage.”

– Azeem Ahmed, Co-Founder of Armor

“This book is a tour de force through all of the important concepts and platforms that you need to know to engage meaningfully with DeFi. I'll be using this as a reference for both myself and newcomers to the space for a good long while.”

— Laurence E. Day, Core Team of Indexed Finance

“The most in-depth and relevant guide to understanding DeFi and all its possibilities”

— DeFi Ted, Advisor of COVER Protocol

CONTENTS

<u>Introduction</u>	1
<u>Part One: State of DeFi</u>	3
<u>Chapter 1: Defi Snapshot</u>	4
<u>DeFi Summer 2020</u>	4
<u>DeFi Ecosystem</u>	7
<u>Rise of Gas Fees</u>	8
<u>DeFi is Going Mainstream</u>	10
<u>Recommended Readings</u>	11
<u>Chapter 2: DeFi Activities</u>	12
<u>Yield Farming</u>	12
<u>Liquidity Mining</u>	13
<u>Airdrops</u>	15
<u>Initial DEX Offerings (IDO)</u>	16
<u>Initial Bonding Curve Offering (IBCO)</u>	16
<u>Liquidity Bootstrapping Pool (LBP)</u>	17
<u>Initial Farm Offering (IFO)</u>	17
<u>Yield Farming: Step-by-Step Guide</u>	18
<u>Associated Risks</u>	24
<u>Conclusion</u>	24
<u>Recommended Readings</u>	25
<u>Part Two: Evaluating DeFi Sectors</u>	26
<u>Chapter 3: Decentralized Exchanges</u>	27
<u>Types of DEX</u>	28
<u>Automated Market Makers (AMMs)</u>	29
<u>What are the existing types of AMMs out there?</u>	31
<u>How are prices determined on a Constant Product AMM?</u>	33
<u>The Various Automated Market Makers (AMMs)</u>	35
<u>Uniswap</u>	35
<u>SushiSwap</u>	37
<u>Balancer</u>	40
<u>Curve Finance</u>	41
<u>Bancor</u>	43
<u>What are the differentiators between the AMMs?</u>	44
<u>I. Pool Fees</u>	44
<u>II. Liquidity Mining</u>	45
<u>III Pool Weightage</u>	46

<u>Associated Risks of using AMMs</u>	47
<u>I. Price Slippage</u>	47
<u>II. Front-running</u>	49
<u>III. Impermanent Loss</u>	51
<u>Notable Mentions</u>	53
<u>Conclusion</u>	53
<u>Recommended Readings</u>	54
Chapter 4: DEX Aggregators	55
<u>DEX Aggregator Protocols</u>	56
<u>1inch Network</u>	56
<u>Matcha</u>	58
<u>Paraswap</u>	59
<u>DEX Aggregator's Performance Factors</u>	60
<u>Which DEX Aggregator offers the most value?</u>	61
<u>Associated Risks</u>	63
<u>Notable Mentions</u>	64
<u>Conclusion</u>	64
<u>Recommended Readings</u>	65
Chapter 5: Decentralized Lending & Borrowing	66
<u>Overview of Lending & Borrowing Protocols</u>	67
<u>Compound</u>	67
<u>Maker</u>	69
<u>Aave</u>	70
<u>Cream Finance</u>	71
<u>Protocols Deep Dive</u>	72
<u>Assets Supported</u>	72
<u>Revenue</u>	73
<u>Total Value Locked (TVL)</u>	74
<u>Utilization ratio (Borrowing Volume / TVL)</u>	75
<u>Lending and Borrowing Rates</u>	76
<u>Lenders</u>	76
<u>Borrowers</u>	77
<u>Associated Risks</u>	78
<u>Notable Mentions</u>	79
<u>Conclusion</u>	80
<u>Recommended Readings</u>	81
Chapter 6: Decentralized Stablecoins and Stableassets	82
<u>Centralized Stablecoin</u>	83
<u>Tether (USDT)</u>	83
<u>Decentralized Stablecoin</u>	84
<u>DAI</u>	84
<u>How do we resolve the stablecoin issue?</u>	85

<u>What are Algorithmic Stablecoins and Stableassets?</u>	86
<u>Rebase Model</u>	87
<u>Ampleforth</u>	88
<u>Seigniorage Model</u>	88
<u>Empty Set Dollar</u>	89
<u>Basis Cash</u>	90
<u>Frax Finance</u>	91
<u>How has Algorithmic Stablecoins fared so far?</u>	92
<u>Why has FRAX succeeded?</u>	93
<u>The Next Generation of Algorithmic Stablecoins and Stableassets</u>	94
<u>Fei Protocol</u>	95
<u>Reflexer</u>	96
<u>Float Protocol</u>	97
<u>How will these new Algorithmic Stablecoins and Stableassets fare?</u>	99
<u>I. Collateralization</u>	99
<u>II. Trader Incentives/Disincentives</u>	100
<u>III. Emergency Powers</u>	100
<u>Associated Risks</u>	101
<u>Notable Mentions</u>	101
<u>Conclusion</u>	102
<u>Recommended Readings</u>	103
<u>Chapter 7: Decentralized Derivatives</u>	104
<u>Decentralized Perpetuals</u>	104
<u>Perpetual Protocol</u>	105
<u>dYdX</u>	106
<u>Comparison between Perpetual Protocol and dYdX (Layer 1)</u>	108
<u>Notable Mentions</u>	109
<u>Decentralized Options</u>	109
<u>Hegic</u>	110
<u>Opyn</u>	111
<u>Comparison between Hegic and Opyn</u>	112
<u>Notable Mentions</u>	113
<u>Synthetic Assets</u>	114
<u>Synthetix</u>	114
<u>UMA</u>	116
<u>Comparison between Synthetic and UMA</u>	118
<u>Notable Mentions</u>	119
<u>Associated Risks</u>	119
<u>Conclusion</u>	120
<u>Recommended Readings</u>	120

<u>Chapter 8: Decentralized Insurance</u>	122
<u>What is Insurance?</u>	122
<u>How does Insurance work?</u>	123
<u>Does Crypto need Insurance?</u>	124
<u>DeFi Insurance Protocols</u>	124
<u>Nexus Mutual</u>	124
<u>Type of Covers</u>	124
<u>1. Protocol Covers</u>	125
<u>2. Custody Covers</u>	125
<u>Cover Purchase</u>	125
<u>Claim Assessment</u>	126
<u>Risk Assessment</u>	126
<u>Token Economics</u>	128
<u>Wrapped NXM (wNXM)</u>	128
<u>Protocol Revenue</u>	129
<u>Armor Protocol</u>	129
<u>arNXM</u>	130
<u>arNFT</u>	130
<u>arCORE</u>	130
<u>arSHIELD</u>	131
<u>Claim</u>	131
<u>Protocol Revenue</u>	131
<u>Cover Protocol</u>	132
<u>Type of Covers</u>	133
<u>Cover Purchase</u>	133
<u>Claim Assessment</u>	134
<u>Risk Assessment</u>	134
<u>Protocol Revenue</u>	135
<u>Comparison between Nexus Mutual and Cover Protocol</u>	135
<u>Capital Efficiency</u>	136
<u>Covers Available</u>	136
<u>Claim Payout Ratio</u>	137
<u>Associated Risks</u>	138
<u>Notable Mentions</u>	138
<u>Conclusion</u>	140
<u>Recommended Readings</u>	141
<u>Part Three: Emerging DeFi Categories</u>	142
<u>Chapter 9: Decentralized Indices</u>	143
<u>DeFi ETF Landscape</u>	144
<u>Index Cooperative (INDEX)</u>	145
<u>Indexed Finance (NDX)</u>	146
<u>PowerPool Concentrated Voting Power (CVP)</u>	147

<u>Comparing the Protocol Indices</u>	147
<u>Protocol Fees</u>	148
<u>Protocol Strategies</u>	149
<u>Fund Weighting</u>	150
<u>Associated Risks</u>	152
<u>Notable Mentions</u>	153
<u>Conclusion</u>	154
<u>Recommended Readings</u>	154
Chapter 10: Decentralized Prediction Markets	155
<u>How do Prediction Protocols work?</u>	156
<u>Market-Making</u>	156
<u>Resolution</u>	157
<u>Prediction Market Protocols</u>	158
<u>Augur</u>	158
<u>Omen</u>	160
<u>What are the other key differences between Augur and Omen?</u>	161
<u>Associated Risks</u>	163
<u>Notable Mentions</u>	163
<u>Conclusion</u>	164
<u>Recommended Readings</u>	165
Chapter 11: Decentralized Fixed-Interest Rate Protocols	166
<u>Overview of Fixed Interest Rates Protocols</u>	168
<u>Yield</u>	168
<u>Saffron.Finance</u>	170
<u>Horizon Finance</u>	171
<u>Which FIRP should I use?</u>	173
<u>Associated Risks</u>	174
<u>Notable Mentions</u>	175
<u>Conclusion</u>	176
<u>Recommended Readings</u>	176
Chapter 12: Decentralized Yield Aggregators	177
<u>Yield Aggregators Protocols</u>	178
<u>Yearn Finance</u>	178
<u>Vaults</u>	179
<u>Strategies</u>	179
<u>Yearn Finance Partnerships</u>	180
<u>Alpha Finance</u>	181
<u>Badger Finance</u>	183
<u>Harvest Finance</u>	184
<u>Comparison of Yield Aggregators</u>	185
<u>Associated Risks</u>	186
<u>Notable Mentions</u>	186

<u>Conclusion</u>	187
<u>Recommended Readings</u>	188
Part Four: Technology Underpinning DeFi	189
Chapter 13: Oracles and Data Aggregators	190
<u>Oracle Protocols</u>	191
<u>Chainlink</u>	191
<u>Band Protocol</u>	193
<u>Data Aggregators</u>	195
<u>The Graph Protocol</u>	195
<u>Covalent</u>	198
<u>Notable mentions</u>	199
<u>Associated Risks</u>	200
<u>Conclusion</u>	200
<u>Recommended Readings</u>	201
Chapter 14: Multi-Chain Protocols & Cross-Chain Bridges	202
<u>Protocols and Bridge Overview</u>	203
<u>Ren Project</u>	203
<u>Lock-and-mint</u>	204
<u>Burn-and-release</u>	204
<u>Burn-and-mint</u>	204
<u>ThorChain</u>	205
<u>Binance Bridge</u>	208
<u>Anyswap</u>	209
<u>Terra Bridge</u>	210
<u>Notable mentions</u>	211
<u>Associated Risks</u>	211
<u>Conclusion</u>	212
<u>Recommended Readings</u>	213
Chapter 15: DeFi Exploits	214
<u>Causes of Exploits</u>	215
<u>1. Economic Exploits/Flash Loans</u>	215
<u>2. Code in Production Culture</u>	215
<u>3. Sloppy Coding and Insufficient Audits</u>	216
<u>4. Rug Pull (Inside Jobs)</u>	216
<u>5. Oracle Attacks</u>	217
<u>6. Metamask Attack</u>	217
<u>Flash Loans</u>	218
<u>What are Flash Loans?</u>	218
<u>Usage of Flash Loans</u>	219
<u>Flash Loan Protocol: Furucombo</u>	220
<u>Case Study: bZx Flash Loans Hack</u>	221

<u>Flash Loan Summary</u>	223
<u>Solutions</u>	224
<u>Internal Insurance Fund</u>	224
<u>Insurance</u>	224
<u>Bug Bounty</u>	225
<u>Other Possible Solutions</u>	225
<u>Tips for Individuals</u>	225
<u>Don't Give Smart Contracts Unlimited Approval</u>	225
<u>Don't Give Smart Contracts Unlimited Approval: Step-by-Step Guide</u>	227
<u>Revoking Unlimited Approvals from Smart Contracts</u>	230
<u>Use a Hardware Wallet</u>	231
<u>Use a Separate Browser Profile</u>	231
<u>Separate Browser Profile: Step-by-Step Guide</u>	232
<u>Conclusion</u>	234
<u>Recommended Readings</u>	235
<u>Chapter 16: The Future of Finance</u>	236
<u>How Long Before Institutions Build on These Networks?</u>	237
<u>Where Does This Take Us in the Next 5 to 10 Years?</u>	238
<u>Closing Remarks</u>	240
<u>Appendix</u>	241
<u>CoinGecko's Recommended DeFi Resources</u>	241
<u>Analytics</u>	241
<u>News Sites</u>	241
<u>Newsletters</u>	242
<u>Podcast</u>	242
<u>Youtube</u>	242
<u>Bankless Level-Up Guide</u>	243
<u>Projects We Like Too</u>	243
<u>Dashboard Interfaces</u>	243
<u>Decentralized Exchanges</u>	243
<u>Exchange Aggregators</u>	243
<u>Lending and Borrowing</u>	243
<u>Oracle and Data Aggregator</u>	244
<u>Prediction Markets</u>	244
<u>Taxes</u>	244
<u>Wallet</u>	244
<u>Yield Optimizers</u>	244
<u>References</u>	245
<u>Chapter 1: DeFi Snapshot</u>	245
<u>Chapter 2: DeFi Activities</u>	246
<u>Chapter 3: Decentralized Exchanges</u>	246

<u>Chapter 4: DEX Aggregators</u>	248
<u>Chapter 5: Decentralized Lending & Borrowing</u>	249
<u>Chapter 6: Decentralized Stablecoins and Stableassets</u>	250
<u>Chapter 7: Decentralized Derivatives</u>	251
<u>Chapter 8: Decentralized Insurance</u>	252
<u>Chapter 9: Decentralized Indices</u>	253
<u>Chapter 10: Decentralized Prediction Markets</u>	254
<u>Chapter 11: Decentralized Fixed-Interest Rate Protocols</u>	255
<u>Chapter 12: Decentralized Yield Aggregators</u>	255
<u>Chapter 13: Oracles and Data Aggregators</u>	256
<u>Chapter 14: Multi-Chain Protocols & Cross-Chain Bridges</u>	256
<u>Chapter 15: DeFi Exploits</u>	257
<u>Chapter 16: DeFi will be the New Normal</u>	257
Glossary	259

INTRODUCTION

When we first wrote the *How to DeFi: Beginner (First Edition)* book in March 2020, we intentionally omitted a lot of information to make it easy for beginners to get started in DeFi. We knew that the book was only scratching the surface of what DeFi has to offer. Of course, it would not appear that way to beginners - many are merely trying to understand a rapidly growing industry infamous for its steep learning curve.

As our readers became familiar with the basics of DeFi, demand for more in-depth knowledge soon followed. We knew that we needed to write a follow-up for readers interested in exploring DeFi further, so we wrote this advanced book to serve this purpose.

We cannot underestimate how easy it is to get lost in DeFi. Every week, new protocol innovations occur, making DeFi more efficient and extending the use-cases further. Fully addressing this knowledge gap is impossible. However, we hope to bridge the gap through our research and analysis of key DeFi segments in this book.

In this book, we will be providing a rundown on the most current state of DeFi and what we can expect in the coming years. We will compare existing protocols under their respective sectors and offer some comparative analysis.

Throughout the book, we will also have **Recommended Readings** at the end of each chapter. In these sections, we will share supplementary reading materials that we believe will be useful as you dive deeper into the DeFi

ecosystem. All credits, of course, go to their respective authors. Kudos to them for making DeFi more accessible!

This book is best for readers who already possess some basic understanding of DeFi and intend to become active users of DeFi. Therefore, we recommend beginners to start with our *How to DeFi: Beginner* book before continuing with *How to DeFi: Advanced*.

We hope that by sharing our learnings, you will have a deeper understanding of DeFi and will be able to decide better which DeFi protocols suit your needs best.

CoinGecko Research Team

Lucius Fang, Benjamin Hor, Erina Azmi, Khor Win Win

1 May 2021

PART ONE: STATE OF DEFI

CHAPTER 1: DEFI SNAPSHOT

The rise of Decentralized Finance (DeFi) took the crypto world by surprise during the summer of 2020, so much that we refer to the period as DeFi Summer 2020. Total Value Locked (TVL), a measure of the amount of capital locked inside DeFi protocols, has been increasing at a breakneck speed, breaching the magic number of \$1 billion in May 2020 and ending the year with \$15.7 billion in TVL.

Since then, DeFi has been growing non-stop, expanding into other non-Ethereum chains. DeFi TVL has reached an astounding figure of \$86.05 billion in April 2021, showcasing the exponential growth of the crypto industry.¹

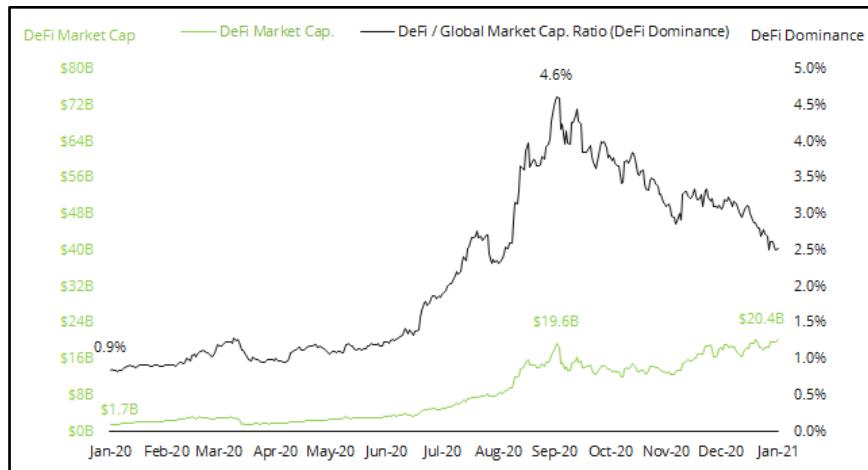
In this chapter, we will zoom into key events of DeFi Summer 2020, take a look at the current DeFi ecosystem, the rise of Ethereum gas fees, and share our thoughts on DeFi becoming mainstream.

DeFi Summer 2020

The crypto space witnessed the meteoric rise of the DeFi sector in 2020, particularly from June to August. The market capitalization of DeFi protocols multiplied 12 times to \$19.6 billion during the peak of the summer,

¹ (n.d.). Defillama - DeFi Dashboard. Retrieved May 4, 2021, from <https://defillama.com/>

now dubbed as DeFi Summer 2020. DeFi dominance, calculated by dividing the DeFi projects' market capitalization over the total crypto market capitalization, rose rapidly from 0.9% to 4.6%.



Source: CoinGecko

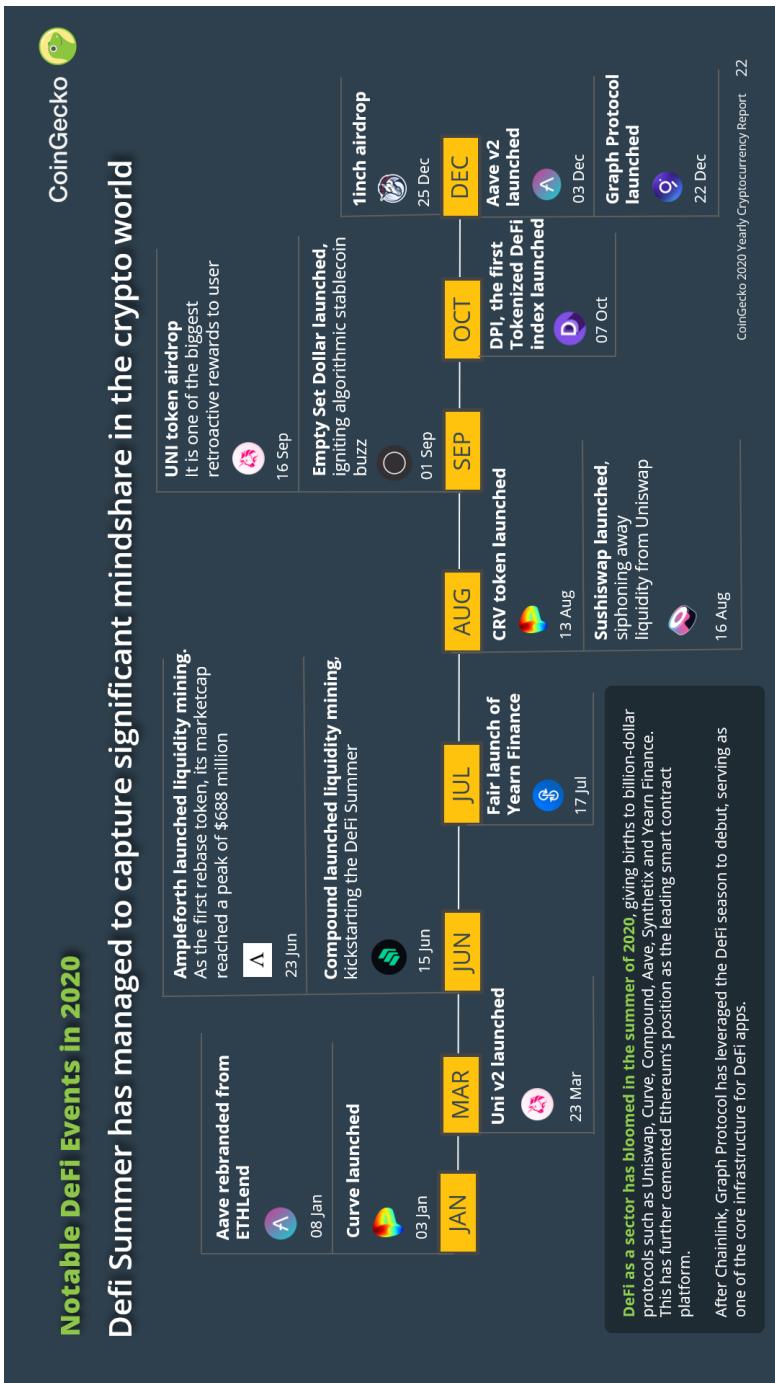
For those of you who were not involved with DeFi in 2020, we have a prepared a short timeline of key DeFi events in 2020 on the next page.

2020 was characterized by the protocol and token launches of many key DeFi projects. Many of these DeFi protocols introduced liquidity mining programs with high yields to attract users to their protocols.

Liquidity mining, a reward program that gives out the protocol's native tokens to users who provide liquidity on the DeFi protocol, is not a foreign concept in DeFi. It was first introduced by Synthetix back in July 2019 and was later popularized by Compound in June 2020.²

The popularity of these liquidity mining programs saw many projects launched in the summer of 2020, with many incorporating food and vegetables token names such as Yam and Pickle. Users had a busy summer being "yield farmers", actively rotating their capital into the various DeFi protocols searching for the highest yields.

² (2019, July 12). Uniswap sETH Pool Liquidity Incentives - Synthetix Blog. Retrieved March 26, 2021, from <https://blog.synthetix.io/uniswap-seth-pool-incentives/>



Yearn Finance, a yield aggregator, kicked off the “fair launch” hype in July 2020. YFI tokens were distributed fairly to anyone who wanted to participate without any private sales to earlier investors. The story continued with the “accidental” launch of the CRV token by Curve Finance in August 2020.

In the same month, SushiSwap, a fork of Uniswap, was launched. SushiSwap conducted a “vampire mining” attack to migrate liquidity away from Uniswap by introducing the SUSHI token to incentivize users.

Not to be outdone, Uniswap did a UNI token airdrop in September 2020, resulting in a windfall profit for all users who interacted with the Uniswap protocol. (Side note: if you had read our *How to DeFi: Beginner* book in 2020 and used Uniswap before the airdrop, you would have received the UNI token too!)

This kickstarted another wave of crypto frenzy, with many projects choosing to launch their tokens to attract more users and growth. Projects without tokens soon find themselves having to consider issuing tokens to compete effectively.

DeFi Ecosystem

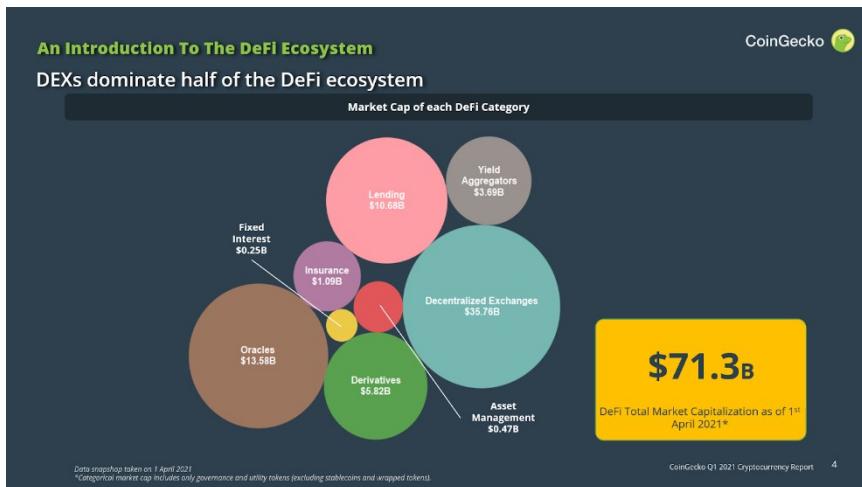
DeFi’s Total Value Locked (TVL) crossed an impressive \$86.05 billion in April 2021. TVL is one of the most widely used metrics in DeFi because it represents the total amount of assets held by each protocol. As a rule of thumb, the more value locked in a protocol, the better it is for the protocol.

In most cases, the locked capital is used to offer services such as market making, lending, asset management, and arbitraging across the ecosystem, earning yields for the capital providers in the process.

However, TVL is not always a reliable metric because its levels can be volatile as capital can be mercenary with temporary incentives such as liquidity mining programs or external catalysts such as smart contract bugs. Hence, it is essential to look at TVL over time to measure the retention of capital and user stickiness.

With such a large amount of capital locked inside the space, various DeFi Dapps have emerged, challenging the norms of financial theories and boundaries. Novel financial experiments are happening daily, giving birth to new categories such as algorithmic stablecoins.

Below is an overview of the burgeoning DeFi ecosystem based on market capitalization. The Decentralized Exchange category is the highest valued category, followed by Oracles and Lending.

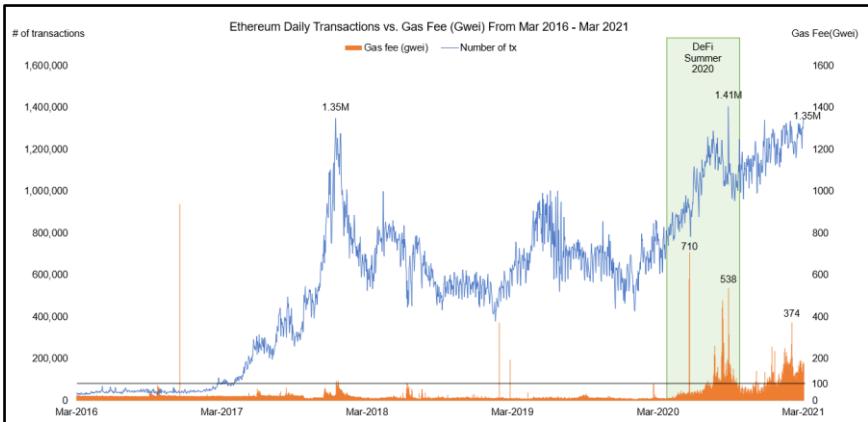


Source: CoinGecko Q1 2021 Report

Rise of Gas Fees

Since the start of 2020, the number of transactions on Ethereum has continuously risen and has exceeded one million transactions per day. The transaction-level seems to be on track to break 2018's peak transaction levels.

DeFi Snapshot



Source: Etherscan

The high number of transactions resulted in an uptrend of gas price, which hit a whopping 700 gwei per transaction by August 2020. Although gas price in 2021 is lower compared to DeFi Summer 2020, the price of ether in 2021 is much higher, resulting in the rise of overall transaction fees. Ethereum broke its previous All-Time High in January 2021 and recorded a new peak of \$4,357 on 12 May 2021!



Source: CoinGecko

The high gas price and the increase in Ethereum price have made many DeFi Dapps on Ethereum no longer economically feasible for users to use without significant capital. Completing a simple swap on Uniswap in Q1 2021 can rack up fees of up to \$100 per transaction, making it only feasible for large swaps. Transaction fees become even higher for more complex transactions such as yield farming transactions.

High transaction fees caused many Ethereum DeFi users to look for cheaper alternatives elsewhere. Some of the other options include moving to Layer-2 (e.g., Optimism, Arbitrum, and zkRollups), sidechains (e.g., xDAI and Polygon), or competing Layer-1 chains (e.g., Binance Smart Chain, Solana, and Terra). We will cover these in more detail in [Chapter 14](#).

DeFi is Going Mainstream

The crypto industry captured a lot of attention in the first half of 2021. Significant events happening globally led to attention-grabbing headlines such as:

1. Tesla's \$1.5 billion initial Bitcoin investment
2. Beeple's \$69 million Non-Fungible Token art sale (*Everydays: the First 5000 Days*) on Christie's
3. Visa supporting USDC as a settlement option on Ethereum
4. Fidelity's plans for a Bitcoin Exchange-Traded Fund
5. Coinbase's listing on the NASDAQ
6. China's positive perception of Bitcoin and other cryptoassets as alternative investments

The increased media attention in the broader crypto market has also led to more eyes on DeFi. In particular, institutional investors started to take notice. For example, in Citibank's Global Perspective and Solutions (Citi GPS) report entitled "Future of Money: Crypto, CBDCs and 21st Century Cash," the 209-year-old lender espoused the benefits of DeFi, including the removal of third-party intermediaries and increased financial transparency.

Notably, the same report also explored various DeFi protocols such as Maker, Compound, Uniswap, and UMA. An in-depth report by the Federal

Reserve Bank of St Louis also highlights DeFi's potential to cause a "paradigm shift in the financial industry and potentially contribute toward a more robust, open, and transparent financial infrastructure".

We also see investment institutions entering the DeFi foray. Grayscale, one of the more famous digital investment funds, is actively offering exposure to DeFi assets (e.g. Chainlink) through share-based trusts. Bitwise Asset Management fund also has a DeFi Index Fund which offers exposure to over 10 DeFi assets such as Aave and Compound. When the fund opened in March 2021, it raised \$32.5M in just two weeks.

DeFi is not stopping at the virtual line either. The envisioned "real world" use-cases of DeFi have already materialized where DeFi protocols are recognized as suitable alternatives from traditional banking instruments.

Centrifuge, one of the first "real world" companies to integrate with MakerDAO, is onboarding non-digital assets as collateral through their app, Tinlake. On 21 April 2021, the company successfully executed its first MakerDAO loan for \$181k with a house as collateral, effectively creating one of the first blockchain-based mortgages.

Recommended Readings

1. Q1 2021 CoinGecko Report
<https://assets.coingecko.com/reports/2021-Q1-Report/CoinGecko-2021-Q1-Report.pdf>
2. Understanding Eth2
<https://ethereum.org/en/eth2/>
3. Beyond Ethereum: Ethereum Killers or Enhancers?
<https://peeman34.medium.com/ethereum-enhancers-or-ethereum-killers-a-new-ecosystem-dd7aeed3d440>

CHAPTER 2: DEFI ACTIVITIES

In this chapter, we will look at the various ways you can participate in the DeFi ecosystem. DeFi protocols democratize access and allow unprecedented freedom for capital providers to offer financial services.

Activities such as market-making, insurance underwriting, and structured product creation were previously only accessible to institutions with a large capital base and specialized knowledge. DeFi has significantly reduced these barriers and availed these activities to the masses.

We will explain further some of the key areas and activities of DeFi, such as yield farming, liquidity mining, airdrops, and Initial DEX Offerings (IDO).

Yield Farming

Yield farming is perhaps one of the most innovative features of DeFi. It refers to the activity of allocating capital to DeFi protocols to earn returns.

Most DeFi protocols are peer-to-peer financial applications where capital allocated is used to provide services to end-users. The fees charged to users are then shared between the capital providers and the protocol. The fees that capital providers get are the intrinsic yield.

In crypto slang, we refer to these investors as “yield farmers” and the yield opportunities as “farms”. Many yield farmers constantly rotate their farms in search of the highest-yielding opportunities.

Below are some examples of yield farming, where the capital provided will be used for a variety of purposes:

- **Exchanges** - Provide capital for market-making on decentralized exchanges, earning transaction fees in return.
- **Lending** - Provide loans to borrowers, earning an interest.
- **Insurance** - Underwrite insurance, earning premiums while undertaking the risk of paying out claims during disasters.
- **Options** - Underwrite options by selling call and put options to earn a yield.
- **Synthetic Assets** - Mint stablecoins or other synthetic assets, earning fees in return.

Liquidity Mining

Financial services is a capital-intensive industry and usually benefits from having economies of scale. This means the more capital that firms have, the better. DeFi protocols are no exception - obtaining sizable capital will be a substantial competitive advantage for DeFi protocols.

In crypto, DeFi protocols incentivize the provision of liquidity via liquidity mining programs. Liquidity mining refers to the reward program of giving out the protocol's native tokens in exchange for capital. These tokens usually come with governance rights and may have the possibility of accruing cash flows from the DeFi protocols.

When designed right, liquidity mining programs are a quick way to bootstrap large sums of liquidity in a short timeframe, albeit with a dilution of token ownership. These programs can also be used to incentivize new users to try out new DeFi protocols. You may think of these incentives similar to how Uber subsidizes rides in the early days using venture capital.

Liquidity mining programs are also a novel way to attract the right kind of community participation for DeFi protocols. As DeFi protocols are open-source in nature, it relies on voluntary contributions from the community. The token distribution encourages community participation in determining the future direction of the protocol.

While it may look like liquidity mining programs offer free rewards, by participating, you are locking up your capital for some time, and your capital has an opportunity cost of earning higher returns elsewhere. Additionally, in many instances, locking up your capital is not risk-free. Yield farming activities typically involve various risks that may lead to loss of money.

The most common form of liquidity mining program is the provision of native token liquidity on decentralized exchanges with base tokens such as ETH, WBTC, or USD stablecoins. Such programs incentivize the creation of liquidity surrounding the protocol's native tokens and enable users to trade these tokens on decentralized exchanges easily.

Example

In August 2020, SushiSwap, a decentralized exchange, launched its governance token named SUSHI and wanted to bootstrap the liquidity between SUSHI and ETH. The team offered to reward free SUSHI tokens to anyone that provides liquidity on the SUSHI/ETH trading pair on SushiSwap.

Users will have to provide both SUSHI and ETH in an equal ratio. Let's assume a user has \$1,000 worth of ETH and would like to participate in this liquidity mining program to earn SUSHI tokens. The user first exchanges half of his ETH into SUSHI. Then he supplies \$500 worth of SUSHI and \$500 worth of ETH into the SUSHI-ETH liquidity pool. He will be given SUSHI-ETH Liquidity Provider (LP) tokens, representing his share in the liquidity pool that can then be staked in the SushiSwap platform to obtain the SUSHI token incentive.

Other types of liquidity mining programs have customizable names based on the intended use case of the capital locked. For example, Compound, a

decentralized lending and borrowing protocol, gives out COMP tokens to lenders and borrowers to incentivize protocol activity. Nexus Mutual, a decentralized insurance protocol, has a Shield Mining program that gives out NXM tokens to NXM token holders who stake their capital on specific protocols to open up more insurance cover capacity. Hegic, a decentralized options protocol, gives rHEGIC tokens to option sellers and buyers to incentivize protocol activity.

There are several websites to monitor yield farming and liquidity mining opportunities:

- <https://www.coingecko.com/en/yield-farming>
- <https://vfat.tools/>
- <https://zapper.fi/farm>

Airdrops

Airdrops are essentially freely distributed tokens. Projects usually conduct airdrops as part of their marketing strategy to generate attention and hype around their token launch, albeit with the tradeoff of diluting token ownership.

Some projects also conduct airdrops to reward early users who have interacted with their protocols. Each protocol will have criteria for qualifying airdrop recipients, such as the timing of interaction and minimum amounts used.

Some notable airdrops are shown below:

Protocol	Token symbol	Date Airdrop	Initial Price	Price as of 1st April 2021	Return
 Uniswap	UNI	16 September 2020	\$3.44	\$28.71	734.59%
 1INCH	1INCH	25 December 2020	\$2.36	\$4.46	88.98%
 PoolTogether	POOL	17 February 2021	\$11.98	\$23.11	92.90%

One of the most notable airdrops was the one conducted by Uniswap. Early users were awarded a minimum of 400 UNI.³ As of 1 April 2021, the airdrop is worth as much as \$11,484. (Side note: if you had read our How to DeFi: Beginner book in 2020 and used Uniswap before the airdrop, you would have received the UNI token airdrop too!)

Initial DEX Offerings (IDO)

Crypto projects have to be creative over their token launch and distribution strategies. With the growing popularity of Decentralized Exchanges (DEX), projects now have a viable option of going direct to users without paying sky-high fees to get listed on centralized exchanges. Crypto project teams can now list their tokens without the need for permission on these DEXs.

However, distributing tokens fairly and to a wide group of users at a fair price is still a difficult endeavor. There are various types of IDOs available, and we will be looking at a few popular types.

Initial Bonding Curve Offering (IBCO)

Initial Bonding Curve Offering, or IBCO, is a fairly new concept meant to prevent front-running practices. Essentially, as more investors provide

³ (2020, September 16). Introducing UNI - Uniswap. Retrieved May 24, 2021, from <https://uniswap.org/blog/uni/>

capital into the bonding curve, the token price will increase from its initial value.

However, it does not matter when you choose to contribute since all investors will pay based on the same final settlement price. Based on the price at the end of the IBCO, each investor will receive a portion of the tokens based on their share of the total capital invested. Projects such as Hegic and Aavegotchi have used this distribution method in their initial token launches with great success.

Liquidity Bootstrapping Pool (LBP)

Hosted using Balancer's Smart Pools, Liquidity Bootstrapping Pools (LBP) is a way for projects to sell tokens using a configurable Automated Market Maker. Usually, these pools would contain the project token and a collateral token, usually denominated in stablecoins. Controllers of the smart pool can change its parameters and introduce a variety of features to the sale, such as a declining price over time as well as pausing any further swaps due to high demand or external vulnerabilities.

Initial Farm Offering (IFO)

First introduced by PancakeSwap, an Initial Farm Offering (IFO) allows users to stake their Liquidity Provider (LP) tokens in exchange for a project's tokens. Using an overflow mechanism, users can stake as much or as little as they want. In the event of oversubscription, any excess tokens are returned to the bidder. IFOs on PancakeSwap use the CAKE-BNB LP tokens, where the project receives BNB tokens in exchange for their newly minted protocol's tokens, while the remaining CAKE tokens are burnt.

Yield Farming: Step-by-Step Guide

In this section, we will be going through a step-by-step guide on yield farming. Using the earlier SUSHI/ETH example in this chapter, we will show how to yield farm using Zapper.

1

The screenshot shows the Zapper.fi dashboard interface. At the top, there are several tabs: Dashboard (highlighted with a red box), Exchange, Pool (highlighted with a red box), Farm, Bridge, and History. Below the tabs, the net worth is displayed as \$2,700.46. Under Account Overview, there are four categories: Wallet (\$2,151.42), Deposits (\$281.84), Staked (\$328.48), and Debt (\$61.27). The 'Pool' tab is currently active, indicating the user is about to start yield farming.

Category	Value
Wallet	\$2,151.42
Deposits	\$281.84
Staked	\$328.48
Debt	\$61.27

Step 1

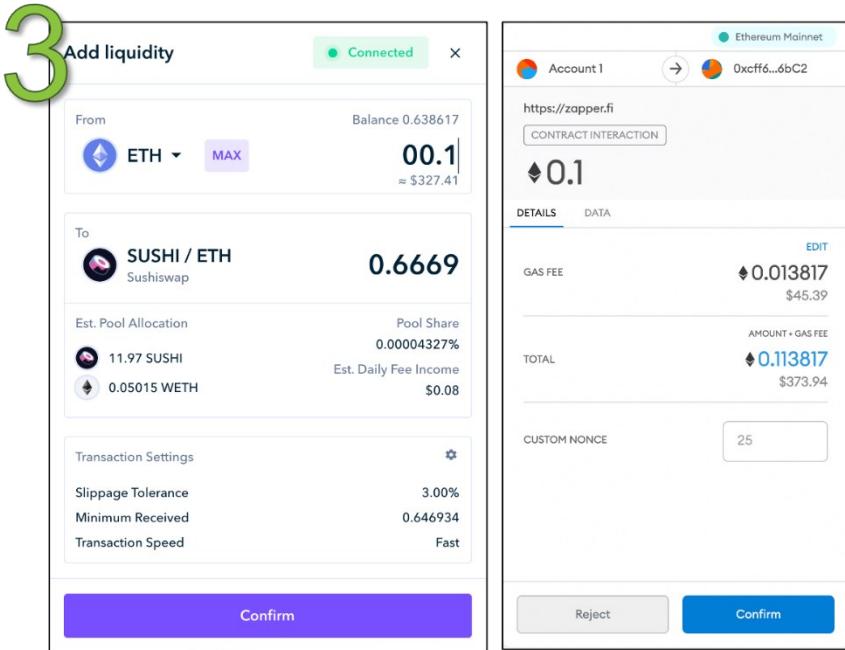
- Connect Ethereum wallet at <https://zapper.fi/dashboard>
- Select the “Pool” tab

2

#	Available Pools	Liquidity	ROI	Invest
1	3Pool Curve Curve	\$1,513,530,625	0.98% (1y) 0.02% (1w)	Invest
2	stETH Curve Curve	\$1,169,379,047	0.04% (1y) 0.00% (1w)	Invest
3	WBTC / ETH SushiSwap	\$948,974,159	12.82% (1y) 0.25% (1w)	Invest
4	sETH Curve Curve	\$934,333,647	0.45% (1y) 0.01% (1w)	Invest
5	FEI / ETH Uniswap V2	\$906,697,368	5.32% (1y) 0.10% (1w)	Invest
6	SUSHI / ETH SushiSwap	\$756,612,959	8.56% (1y) 0.16% (1w)	Invest

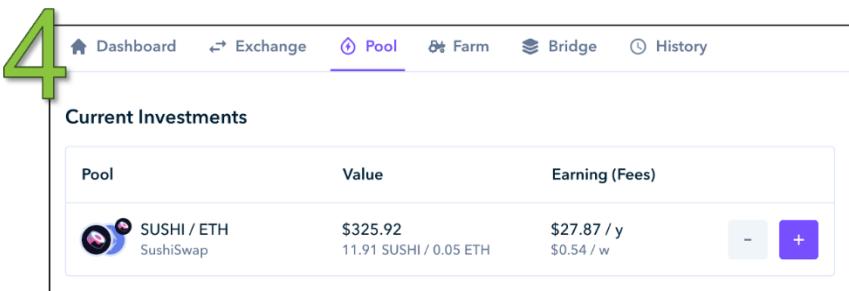
Step 2

- In this guide, we are going to yield farm SUSHI/ETH to get the SUSHI token.
- Click “Invest”



Step 3

- In Zapper, we can swap any single asset to any LP token. Here we will swap ETH to SUSHI/ETH.
- Confirm the transaction.



Step 4

- The SUSHI/ETH position will appear under the Current Investments section.

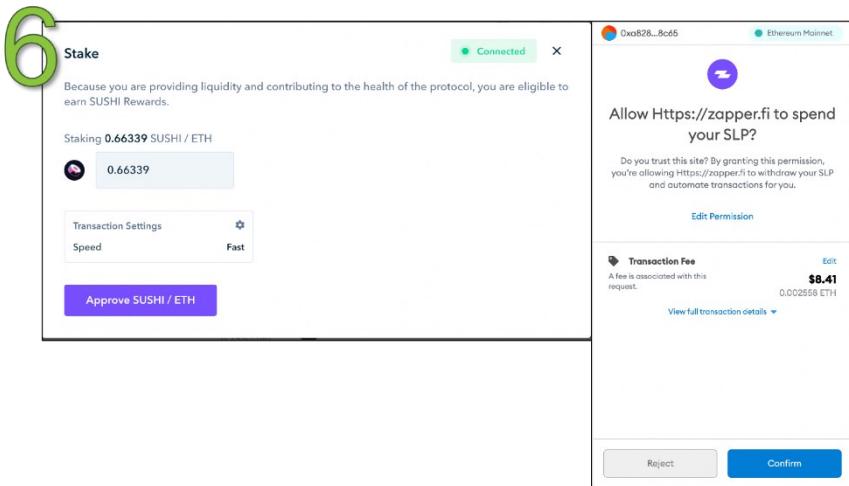
5

#	Assets	Liquidity	ROI (1y)	Rewards
1	stETH Curve Curve	\$1,169,379,046.99	2.76% (1y) 0.05% (1w)	
2	sETH Curve Curve	\$931,818,683.80	8.28% (1y) 0.15% (1w)	
3	WBTC / ETH SushiSwap	\$887,201,167.16	13.22% (1y) 0.25% (1w)	
4	3Pool Curve Curve	\$746,131,297.57	6.49% (1y) 0.12% (1w)	
5	SUSHI / ETH SushiSwap	\$739,208,532.84	22.75% (1y) 0.43% (1w)	Stake
6	HBTC Curve Curve	\$528,552,733.45	6.54% (1y) 0.12% (1w)	

Step 5

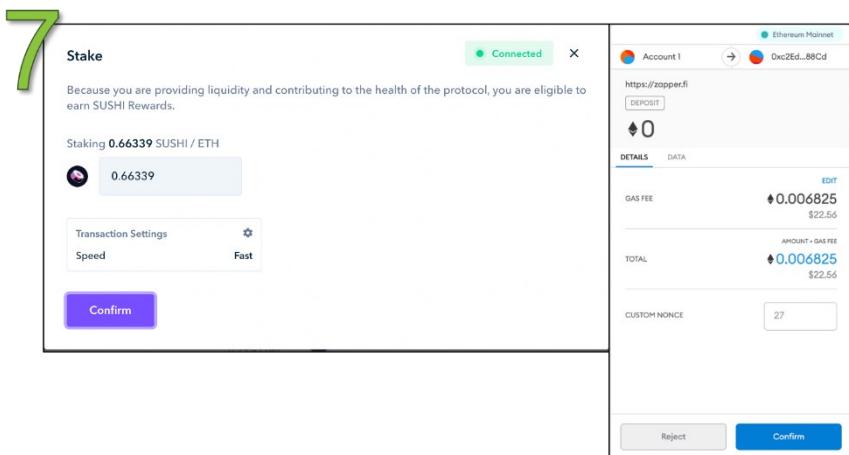
- The “Farm” tab lists yield farming opportunities and their respective expected returns.
- A green “Stake” button will appear if we have the underlying asset for the yield farming opportunity.
- Click “Stake”

How to DeFi: Advanced



Step 6

- Approve the transaction. This allows Zapper access to our SUSHI/ETH LP token.



Step 7

- Confirm the transaction.

8

Assets	Value	Earning
SUSHI / ETH SushiSwap	\$327.62 0.6633 tokens	\$27.91 / yr \$0.54 / w

Claim Unstake

Step 8

- Now we will be able to earn the trading fees and the liquidity mining reward.
- Click “Claim” to see the rewards.

9

Claim your rewards

Connected X

Claiming 0.000190368 SUSHI Rewards

0.000190368

Transaction Settings Speed Fast

Confirm

Step 9

- The rewards can be claimed by clicking “Confirm”.
- We can exit the position by clicking “Unstake”, as shown in the image at step 8.

Associated Risks

Once you are familiar with the DeFi ecosystem, you will inevitably see various protocols offering eye-popping yields, sometimes more than 1,000% Annual Percentage Yield (APY)! While it is tempting to plow your money in, such APYs are usually temporary and will eventually stabilize to lower numbers once other yield farmers start coming in.

Given the fast-paced world of DeFi, investors also have to make quick decisions on whether a project is worth investing in. Fear of Missing Out (FOMO) is real and should not be taken lightly.

Regardless of whether you are a trader, investor, or yield farmer, one should always be wary of the usual risks such as smart contract risk, impermanent loss risk, and relevant systemic risk. No matter how technically sound a project may be, liquidity exploits are always possible from malicious actors.

Users need to understand that the DeFi ecosystem is still nascent, and most DeFi activities are still experimental. We will be covering the different types of risks associated with each DeFi category throughout this book, and dedicate the whole of [Chapter 15](#) to risks associated with smart contract exploits.

Conclusion

DeFi is groundbreaking. We are witnessing a financial revolution happening right in front of us, one which democratizes access to finance, promotes financial inclusion, and promises financial transparency. Although DeFi in its current iteration is not perfect, it does provide us with a glimpse of what the future may look like.

Anyone in the world with access to the Internet can now participate in this grand financial experiment. Crypto financial primitives such as being a liquidity provider and the tokenization of ownership allow new forms of organizations to be formed. It will not be long before we see DeFi protocols being more valuable than the largest companies in the world.

Recommended Readings

1. Governance Tokens: Investing in the Building Blocks of a New Economy
<https://thedefiant.io/governance-tokens-investing-in-the-building-blocks-of-a-new-economy/>
2. New Models for Utility Tokens
<https://multicoin.capital/2018/02/13/new-models-utility-tokens/>
3. Liquidity Bootstrapping FAQ
<https://docs.balancer.finance/smart-contracts/smart-pools/liquidity-bootstrapping-faq>
4. What is Yield Farming
<https://learn.zapper.fi/articles/what-is-yield-farming>

PART TWO: EVALUATING DEFI SECTORS

CHAPTER 3: DECENTRALIZED EXCHANGES

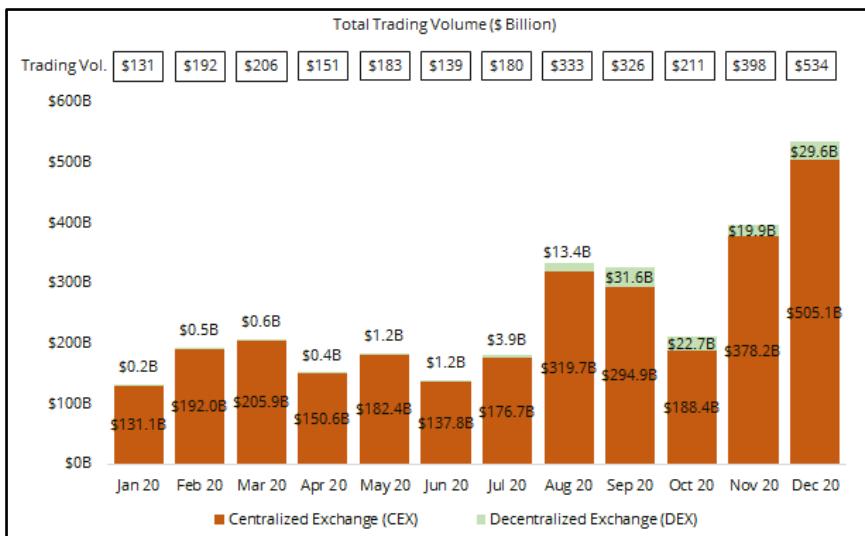
Whether you are trying to make a simple swap or actively trade, you will require the services of an exchange. Ideally, the exchange must have low latency and deep liquidity so that you have the best price execution and do not get impacted by price slippage.

Historically, Centralized Exchanges (CEXs) have provided better liquidity and have facilitated most large trades. However, they carry several weak points - the most notable being users of centralized entities do not hold custody of their assets. For example, in September 2020, KuCoin suffered a \$281 million hack after a security breach.⁴ CEXs could also halt trading and block users from withdrawing their funds at any time.

In 2020 and 2021, Decentralized Exchanges (DEXs) have grown rapidly and have started to rival their centralized counterparts. The Top-9 DEX-CEX ratio had improved from only 0.2% in January 2020 to 5.9% in December 2020. In 2020, top-9 DEXs recorded an exponential 17,989% growth to \$30 billion in trading volume.⁵

⁴ (2020, September 25). Over \$280M Drained in KuCoin Crypto Exchange Hack - CoinDesk. Retrieved March 13, 2021, from <https://www.coindesk.com/hackers-drain-kucoin-crypto-exchanges-funds>

⁵ “Q1 2021 Quarterly Cryptocurrency Report - CoinGecko.” 15 Apr. 2021, <https://www.coingecko.com/buzz/q1-2021-quarterly-cryptocurrency-report>. Accessed 20 Apr. 2021.



Source: CoinGecko 2020 Yearly Report

But what exactly makes a DEX...a DEX?

A DEX is a platform that enables trading and direct swapping of tokens without the need for an intermediary (i.e., centralized exchange). You do not need to go through the hassles of Know Your Customer (KYC) processes nor are you subjected to jurisdictional limits.

Types of DEX

There are two types of DEXs:

1. Order Book Based-DEXs

An order book is a list of buy and sell orders for a particular asset at various price levels.

Order book-based DEXs like dYdX, Deversifi, and Loopring operate similarly to CEXs where users can set buy and sell orders at either their chosen limit prices or at market prices. The main difference is that in CEXs, assets for the trade are held on the

exchanges' wallets, whereas for DEXs, assets for the trade are held on users' wallets.

Order books for DEXs can either be on-chain or off-chain. On-chain order book-based DEXs have all orders recorded on the blockchain. However, this is no longer feasible on Ethereum due to high gas prices. That said, this is still doable on Ethereum Layer 2 solutions like xDai or high-throughput Layer 1 blockchains like Solana.

Off-chain order book-based DEXs have trade orders recorded outside the blockchain. The trade orders remain off-chain until they are matched, at which point the trades are executed on-chain. Although this approach has lower latency, some may argue that DEXs that utilize this method are considered semi-decentralized.

2. Liquidity Pool Based-DEXs

Liquidity pools are token reserves that sit on DEXs' smart contracts and are available for users to exchange tokens with. Most liquidity pool-based DEXs make use of Automated Market Makers (AMM), a mathematical function that predefines asset prices algorithmically.

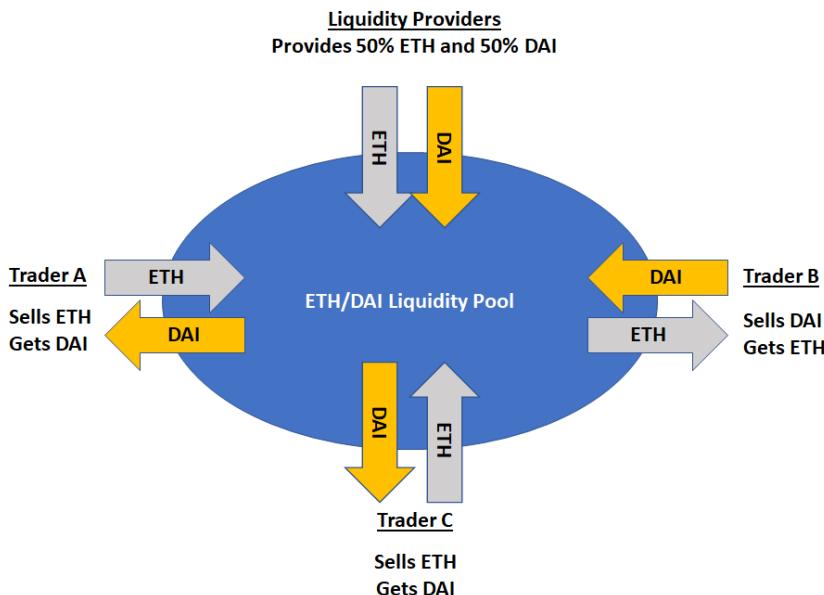
AMM is one of the most innovative inventions from DeFi in recent years. It enables 24/7 market hours, higher capital accessibility, and efficiency. There are various types of AMMs, and different DEXs have implemented the various 'flavors'. The majority of DEXs that launched during the DeFi summer 2020 are AMM-based DEXs such as Uniswap, SushiSwap, Curve, Balancer, and Bancor.

Since many of the newer DEXs are AMM-based, we will focus the rest of this chapter going through a few examples of AMMs.

Automated Market Makers (AMMs)

In our *How to DeFi: Beginner* book, we have gone through Uniswap, the most popular AMM. Here is a recap of how liquidity pools in AMMs work.

Unlike centralized exchanges, which have bids and ask orders placed on order books, AMMs do not have any order books. Instead, it relies on liquidity pools. Liquidity pools are essentially reserves that hold two or more tokens that reside on a DEX's smart contract that are made readily available for users to trade against.



You can think of liquidity pools as just pools of tokens that you can trade against. If you wish to swap ETH to DAI, you will trade on the ETH/DAI liquidity pool by adding ETH and removing an amount of DAI determined algorithmically from the liquidity pool.

Depositors, known as Liquidity Providers (LPs), seed these liquidity pools. LPs deposit their tokens into the liquidity pool based on the predefined token weights for each AMM (in Uniswap's case - 50% for each token).

LPs provide funds in liquidity pools because they can earn a yield on their funds, collected from trading fees charged to users trading on the DEX. Anyone can become an LP and automatically market-make a trading pair by depositing their funds into the smart contract.

With AMMs, traders can have their orders executed seamlessly without the need for a centralized market maker providing liquidity on a centralized exchange like Coinbase or Binance. Instead, orders are executed automatically via a smart contract that will calculate trade prices algorithmically, including any slippage from the trade execution. You may thus consider order book-based exchanges as following the peer-to-peer model while AMMs follow the peer-to-contract model.

What are the existing types of AMMs out there?

AMMs are a mathematical function to price assets algorithmically based on liquidity pools. Currently, several AMM formulas are utilized to cater to different asset pricing strategies.

Some of the more popular AMM formulas are as follow:

I. Constant Product Market Makers

$$x * y = k$$

The Constant Product Market Maker formula was first popularized by Uniswap and Bancor and the most popular AMM in the market. When plotted, it is a convex curve where x and y represent the quantity of two tokens in a liquidity pool, and k represents the product. The formula helps create a range of prices for the two tokens depending on each token's available quantities.

To maintain k as constant, when the supply of x increases, the supply of y must decrease, and vice-versa. Therefore, the resulting price is inherently unstable as the size of trades may affect the price in relation to pool size. Impermanent loss may occur by higher slippage caused by large trades.

II. Constant Sum Market Maker

$$x + y = k$$

The Constant Sum Market Maker formula creates a straight line when plotted. It is an ideal model for zero slippage trade but, unfortunately does not offer infinite liquidity. This model is flawed

as it presents an arbitrage opportunity when the quoted price is different from the market price of the asset traded elsewhere. Arbitrageurs can drain the entire reserves in the liquidity pools, leaving no more available liquidity for other traders. This model is unfit for most AMM use cases.

III. Constant Mean Market Maker

$$v = \prod_t B_t^{w_t}$$

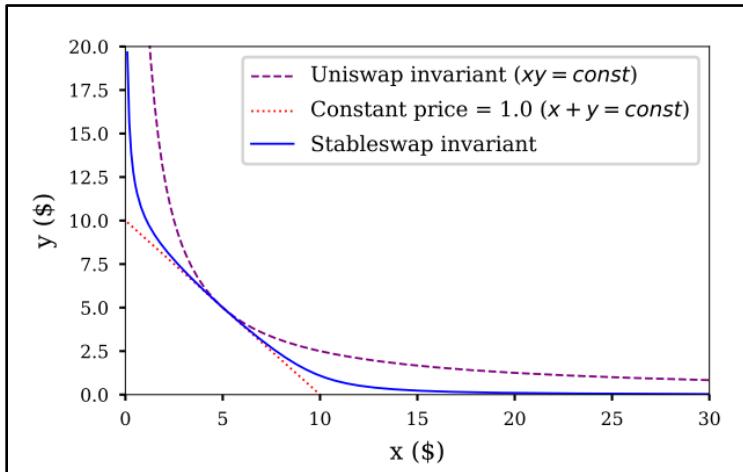
The Constant Mean Market Maker formula, or also known as Value Function, was made popular by Balancer. It allows for liquidity pools with more than two tokens and different token ratios beyond the standard 50/50 distribution. Rather than the product, the weighted geometric mean remains constant. This allows for variable exposure to different assets in the pool and enables swaps between any of the liquidity pool's assets.

IV. Stableswap Invariant

$$An^n \sum X_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i}$$

The StableSwap Invariant formula is a hybrid of the Constant Product and Constant Sum formula. It was made popular by Curve Finance.

Trading occurs on a Constant Sum curve when the portfolio is relatively balanced and switches to a Constant Product curve when imbalanced. This allows for lower slippage and Impermanent Loss but is only applicable to assets with a similar value as the price of the desired trading range is always close to 1. For example, this will be useful for trading between stablecoins (DAI and USDC) and wrapped assets (wBTC and sBTC).



Source: <https://curve.fi/files/stableswap-paper.pdf>

This graph shows the Constant Product Market Maker (purple line) and Constant Sum (red line) curves with a Stableswap Invariant hybrid curve used by Curve Finance (blue line) in the middle. We can see that the Stableswap Invariant curve creates deeper liquidity near the Constant Sum curve. The result is a line that returns a linear exchange rate for most trades and exponential prices for larger trades.

How are prices determined on a Constant Product AMM?

Let's look at a simple example of a Constant Product Market Maker and see how asset prices are determined algorithmically. It works by maintaining a constant product formula based on the amount of liquidity available for each asset in the pool.

To see how it works via the popular AMM, we will look at constant product market makers that Uniswap popularized:

It is important to note that the market price on AMMs changes only when the reserve ratio in the pool changes. Thus, an asset price on AMM might differ from other exchanges.

Example

Based on the Constant Product Market Maker formula

$$x * y = k$$

x = the reserve token x

y = the reserve token y

k = the constant total liquidity that determines the price of tokens in the liquidity pool

For example:

There are 61,404,818 DAI and 26,832 ETH in Uniswap's DAI/ETH liquidity pool as of 21 April 2021. The reserve ratio implies that ETH's price at the time of writing is $61,404,818 \text{ DAI} / 26,832 \text{ ETH} = 2,289 \text{ DAI}$.

Assuming 1 ETH is now valued at 2,289 DAI on Uniswap. But when the price of ETH drops to 2,100 DAI elsewhere, such as on Balancer, an arbitrage opportunity presents itself. Arbitrageurs will take advantage of the price differences by buying cheap ETH on Balancer and selling it off on Uniswap for a quick profit (ignoring the trading fee for simplicity). Arbitrageurs will repeat this until the price reaches equilibrium between the two exchanges.

The Various Automated Market Makers (AMMs)

Uniswap



Uniswap is a decentralized exchange protocol on Ethereum that allows direct token swaps without giving up the custody of your funds. To use Uniswap, all you need to do is send your tokens from your wallet to Uniswap's smart contract, and you will receive your desired tokens in return in your wallet.

Uniswap's journey began in November 2018 when it launched its first iteration, Uniswap version 1. It is one of the first AMM-based DEX that popularized Constant Product Market Makers formula:

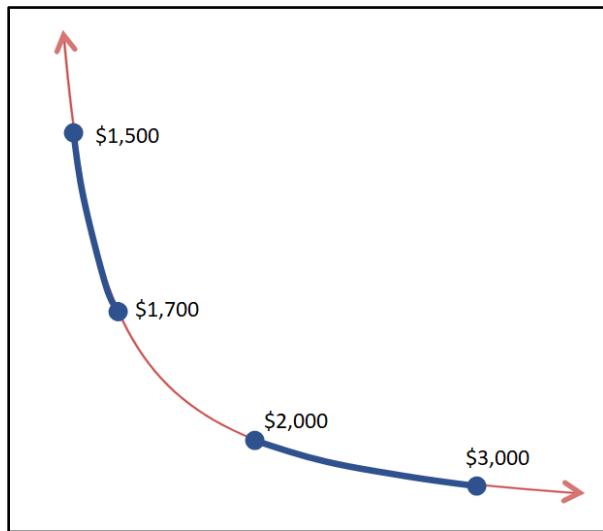
$$x * y = k$$

In May 2020, Uniswap upgraded its smart contract to Uniswap version 2 with added features. The new version expanded trading pairs to support any ERC-20 tokens.

On 5 May 2021, Uniswap released the latest iteration, Uniswap version 3. In the latest iteration, Uniswap introduced two main new features:

1. Concentrated liquidity

With Uniswap version 3, LPs can control the price ranges where they would like to provide liquidity. For example, an LP for the ETH/DAI liquidity pool may choose to allocate 30% of his capital to the \$2,000 - \$3,000 price range and the remaining 70% to the \$1,500 - \$1,700 price range.



The new active management of liquidity on Uniswap version 3 now results in higher capital efficiency for LPs. A by-product of this is that LPs will receive Non-Fungible Tokens (NFTs) instead of fungible ERC-20 tokens representing their LP positions.

2. Multiple pool fee tiers

Uniswap version 3 offers a three-tier pool fee that Liquidity Providers can choose accordingly:

- a. 0.05%
- b. 0.30%
- c. 1.00%

For example, the USDC/DAI trading pair has low price volatility and may warrant a lower 0.05% pool fee. The ETH/DAI trading pair has higher price volatility and would warrant a 0.30% pool fee. Meanwhile, the 1.00% pool fee may be more appropriate for more long-tail or exotic trading pairs.

SushiSwap



SushiSwap was launched on 28 August 2020 by a pseudonymous developer known as Chef Nomi. It was a fork of Uniswap's version 2 source code and utilized the same Constant Product Market Maker model. SushiSwap introduced a SUSHI token at a time when Uniswap did not yet have its UNI token. The attractive yield farming rewards offered by SushiSwap caught the attention of many people in the crypto community.

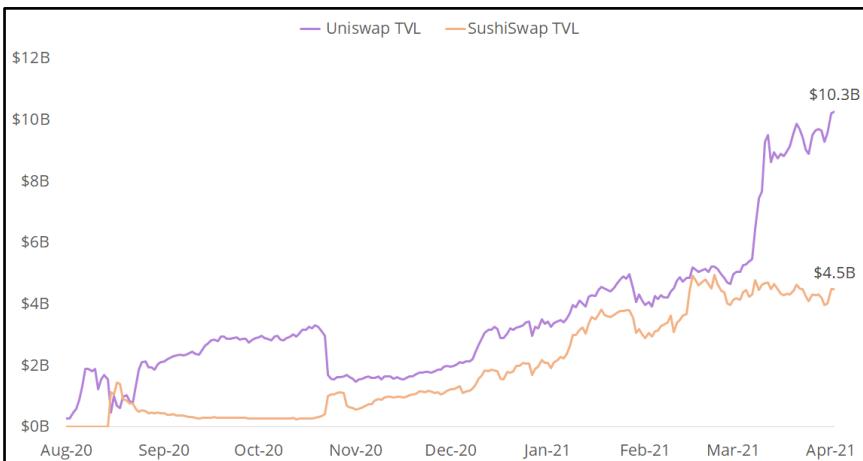
On 9 September 2020, SushiSwap launched a “vampire attack” on Uniswap’s liquidity whereby anyone staking their Uniswap LP tokens on SushiSwap will have their underlying liquidity on Uniswap migrated over to SushiSwap. This attack drained over half of Uniswap’s liquidity and saw its Total Value Locked (TVL) went from \$1.55 billion to \$470 million. Simultaneously, SushiSwap’s TVL increased to \$1.13 billion overnight.

Despite the “vampire attack”, Uniswap has stayed resilient and recovered its TVL lead over SushiSwap very quickly. As of April 2021, SushiSwap now has \$4.5 billion in TVL, slightly half of Uniswap’s \$10.3 billion in TVL.

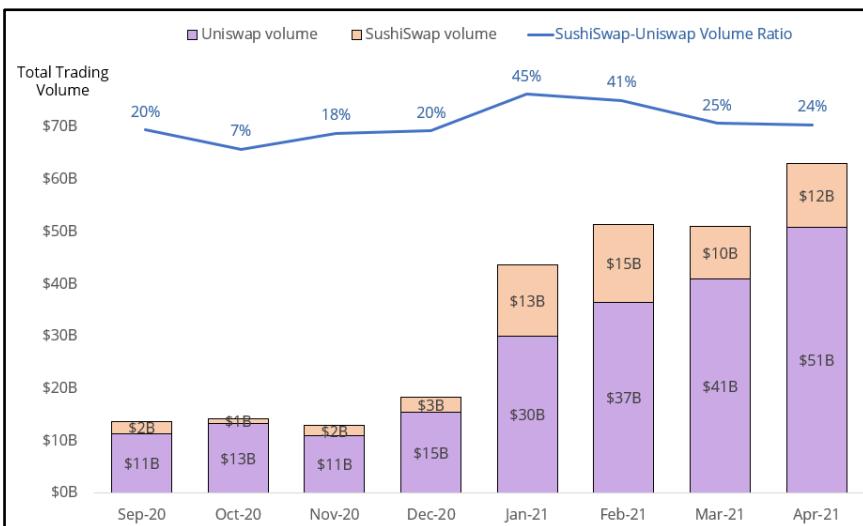
SushiSwap has grown significantly and is now the second-largest DEX behind Uniswap. As of March 2021, Uniswap’s trading volume is four times higher than SushiSwap’s, signifying the substantial lead that Uniswap has in the DEX market. In the first two months of 2021, SushiSwap performed strongly and had 45% of Uniswap’s trading volume.

Since its launch, SushiSwap has differentiated itself by offering a more comprehensive product range. It has also partnered (merged) with Yearn Finance, a yield farming aggregator protocol, and is now the AMM arm of

Yearn Finance.⁶ The key difference between the two lies in the pool fees, available trading pairs, and supported blockchains.



Source: DeBank



Source: CoinGecko

⁶ “Yearn x Sushi 行ってきます . I had been outspoken in the ... - Medium.” 30 Nov. 2020, <https://medium.com/icarn/yearn-x-sushi-%E8%A1%8C%E3%81%A3%E3%81%A6%E3%81%8D%E3%81%BE%E3%81%99-41b2f78b62e9>. Accessed 11 May. 2021.

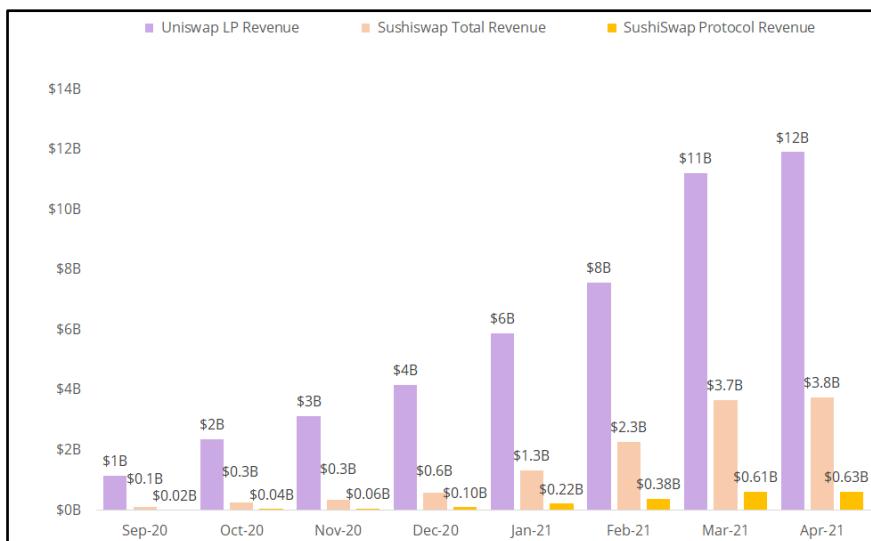
	Uniswap	SushiSwap
Type of fees	Fixed	Fixed
Pool fees	0.30%	0.30%
Protocol	0.00%	0.05%
Liquidity Providers	0.30%	0.25%

*Pool fees as of 1st April 2021

From the table above, we can see that both Uniswap and SushiSwap have a trading fee of 0.3%. However, on SushiSwap, 0.05% of the trading fee goes to the protocol, which is then distributed to SUSHI token holders.

Uniswap does not currently distribute fees to UNI token holders, although this can be activated by UNI governance voting. As of April 2021, Uniswap's LPs receive a higher share of revenue (0.30%) compared to SushiSwap's LPs (0.25%).

We can see from the chart below that both Uniswap and Sushiswap's protocol revenues have grown significantly over the past year.

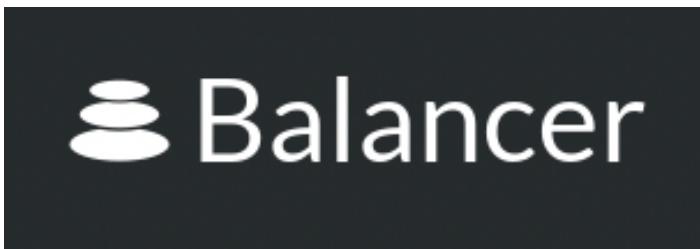


Source: TokenTerminal

Uniswap also has more than 2,000 supported trading pairs, which is approximately five times higher than SushiSwap - this suggests that there are more long-tail tokens supported and traded on Uniswap.

Uniswap currently only supports trading on Ethereum with plans to move to Layer 2 via Optimism. In contrast, SushiSwap operates on nine different blockchains, namely Ethereum, Binance Smart Chain, Polygon, Fantom, Huobi Ecosystem, xDAI, Harmony, Avalanche and OKExChain.

Balancer



Balancer positions itself as a portfolio manager in addition to its AMM-based DEX. Instead of paying fees to invest in a fund, Balancer pool holders collect fees from traders that arbitrage liquidity pools. This essentially creates an index fund that gets paid when the fund is rebalanced, adding another source of income for liquidity providers.

Unlike Uniswap that supports only two assets, Balancer supports multi-asset pools. Pool creators are also allowed to set customized fees ranging from 0.00001% to 10%. This flexibility opens up more possibilities to pool creations.

There are three types of liquidity pools:

1. **Public pool** - Anyone can add liquidity, but the pool parameters are permanently fixed. This is the most trustless pool.
2. **Private pool** - Flexible parameters. The owner is the only entity that can change the parameters and add liquidity. This makes the pool custodial and centralized.

3. **Smart pool** - Anyone can add liquidity. The pool supports fixed and dynamic parameters, which can be changed on an ongoing basis. This is the most flexible pool.

Balancer version 2 supports up to 16 different assets in a pool and allows for the creation of smart pools. Smart pools are especially useful for treasury management, where the pools can act as an automatic token buyback machine.⁷ Plus, it also allows for idle assets in the pools to be lent out to lending protocols, improving the pools' yields.

Balancer has also introduced an innovative Initial DEX Offerings (IDO) method called Liquidity Bootstrapping Pools (LBPs). They are short-lived smart pools with dynamic weighting over time. Token price is set at a high price and is expected to fall throughout the sale. Whales and bots are disincentivized to buy the tokens all at the start, allowing for a more democratic way of fundraising.

Curve Finance



Curve Finance is an AMM-based DEX with the main focus of facilitating swaps between assets of similar value. This is useful in the DeFi ecosystem as there are plenty of wrapped and synthetic tokens that aim to mimic the price of the underlying asset. Curve Finance currently supports USD stablecoins, EUR stablecoins, wrapped/synthetic BTC, and wrapped/synthetic ETH assets.

⁷ “Placeholder VC: Stop Burning Tokens – Buyback And Make Instead.” 17 Sep. 2020, <https://www.placeholder.vc/blog/2020/9/17/stop-burning-tokens-buyback-and-make-instead>. Accessed 11 May. 2021.

For example, one of the biggest liquidity pools is 3CRV, a stablecoin pool consisting of DAI, USDT, and USDC. The ratio of the three stablecoins in the pool is based on the supply and demand of the market. Depositing a coin with a lesser ratio will yield the user a higher percentage of the pool. When the ratio is heavily tilted to one of the coins, it may serve as a good chance to arbitrage.

Curve Finance also supports yield-bearing tokens on Compound, Aave, and Yearn Finance. Curve collaborated with Yearn Finance to release the yUSD pool that consists of yield-bearing token yDAI, yUSDT, yUSDC, and yTUSD. Users who participated in this pool will have yield from the underlying yield-bearing tokens, swap fees generated by the Curve pool, and CRV liquidity mining rewards offered by Curve Finance. Liquidity providers of this pool are able to earn from three sources of yield.

To promote the liquidity of more long-tail tokens, Curve introduced the concept of base pools and metapools. A metapool is a single token pool with another base pool that allows users to trade the single token seamlessly. Currently, the most liquid base pool is the 3CRV pool.

For example, there is a metapool with UST (a USD stablecoin issued on the Terra blockchain) with the 3CRV base pool. Users can trade UST and the three USD stablecoins in the 3CRV pool. By separating the base pool and metapool, Curve is able to separate UST's systemic risks from 3CRV's liquidity pool.

The creation of metapools help Curve in the following ways:

- Prevents dilution of existing pools
- Allows Curve to list illiquid assets
- Higher volume and trading fees for CRV token holders

Bancor



Bancor Network

Launched in 2017, Bancor was one of the first AMM-based DEX. Bancor utilizes a modified Constant Product Market Maker curve, similar to Uniswap. Bancor's approach to this model differs from the arbitrary two-asset curve formula used by Uniswap.

Instead of pairing a base token to any target ERC-20 token like Uniswap, Bancor uses its native token, Bancor Network Token (BNT), as the intermediate currency. There are separate pools for each token traded against BNT.

Bancor version 2 introduces several innovations such as single-sided staking and impermanent loss insurance.

Most AMMs require LPs to provide an equal ratio of each asset represented in the pool. This brings inconvenience to LPs who may only want exposure to a single asset. Bancor version 2 allows LPs to contribute a single asset and maintain 100% exposure on it. LPs can stay long on a single asset with single-sided liquidity while earning swap fees and liquidity mining rewards.

Impermanent loss is a risk that concerns most LPs on AMMs. Bancor incentivizes liquidity by offering compensation for any impermanent loss to LPs. Currently, the payout increases by 1% each day and reaches 100% after 100 days. This impermanent loss coverage encourages LPs to stay in the liquidity pool for at least 100 days. There is a 30-days cliff before impermanent loss protection kicks in.

Bancor also introduced vBNT and Vortex to improve the use cases of BNT tokens. Users receive vBNT when staking BNT in a whitelisted Bancor pool. The exchange rate of BNT to vBNT is 1:1. vBNT can be used for several functions:

- Vote in Bancor governance
- Stake in vBNT/BNT pool for swap fees
- Borrow other tokens on Bancor by using vBNT as collateral (Vortex)

Vortex allows BNT holders to borrow against their staked BNT. The proceeds can be used for leverage or any other purpose, increasing the capital efficiency of holding BNT.

What are the differentiators between the AMMs?

Now that you are familiar with the various AMMs in the market, let's look at three features that make each of them distinct. For simplicity's sake, we will focus on Uniswap v2, Curve, Balancer, and Bancor.

I. Pool Fees

To incentivize users to add liquidity, DEXs allow LPs to earn trading fees on their platform. The fees help LPs with price fluctuations and impermanent loss risks.

Below is a summary of pool fees for the four DEXs in April 2021:

	Uniswap	Curve	Balancer	Bancor
Type of fees	Fixed	Fixed	Variable	Variable
Pool Fees	0.30%	0.04%	Between 0.0001% and 10%	Up to 5.00%*
Protocols	0.00%	0.02%	Depending on each pool and it can be zero.	0.00%**
Liquidity providers	0.30%	0.02%	Between 0.0001% and 10%	Up to 5.00%*

Source: Uniswap, Curve, Balancer, and Bancor.

* The trading fee is controlled by the pool creator. The highest fee as of 26th April 2021 was 5%

** The accrued trading fee that goes to the protocol is used as impermanent loss insurance and not as revenue. It will be burnt once withdrawn from the pool.

Uniswap and Curve implemented a fixed trading fee for every swap made on their platforms. The main difference lies with the split -

Uniswap provides the entire trading fee to LPs while Curve splits the trading fee equally between the protocol and LPs.

For Balancer and Bancor, the trading fees are variable and controlled by the pool's creator.

II. Liquidity Mining

We have touched on this concept in [Chapter 2](#). Briefly, liquidity mining refers to the process of providing liquidity to a protocol, and in return rewarded for the protocol's native tokens.

It is one of the most popular ways to bootstrap liquidity on a DEX and compensate liquidity providers for undertaking the impermanent loss risk.

Each of the four DEXs has its own native token:

Protocol	Coin name	Ticker
	Uniswap	UNI
	Curve Dao Token	CRV
	Balancer	BAL
	Bancor Network Token	BNT

As of 1st April 2021, Uniswap is the only DEX without an active liquidity mining program out of the four DEXs.

III. Pool Weightage

Most AMMs such as Uniswap and Bancor have a standard 50/50 pool weightage whereby liquidity providers must supply an equal value of the two tokens. However, Balancer has a variable pool supply criteria and Curve has a dynamic pool supply criteria.

On Balancer, users can set variable weights for each pool. Pools are constantly rebalanced to ensure they follow the variable weights set. For example, an 80/20 BAL/WETH pool on Balancer means that you will have to split your capital to 80% BAL tokens and 20% WETH tokens when supplying liquidity to the pool.



Source: <https://pools.balancer.exchange/#/>

On Curve, the pool weightage is dynamic and will change according to the reserve size. Unlike the other AMMs, Curve does not rebalance its pools or try to keep them in a balanced ratio.

Let's look at an example of the 3CRV pool consisting of DAI, USDC, and USDT. Ideally, this pool is equally weighted between the three stablecoins. However, the snapshot below had USDC with the highest weightage (41.98%) and DAI with the lowest weightage (24.80%). If you are interested in providing liquidity to this pool, you do not need all three tokens but simply contribute any of the three tokens to the pool. By doing so, you will alter the pool supply weight dynamically.



Source: <https://curve.fi/3pool> as of 30th April 2021

Associated Risks of using AMMs

Using an AMM-based exchange does not come without risks. Below we outline three risks both from a Trader and Liquidity Provider perspectives.

I. Price Slippage

Based on the AMM formulas, the quoted price is dependent on the ratio of the token reserves.

In the Constant Product Market Maker formula ($x * y = k$), the larger the order, the larger the price slippage that a user will incur. This is subjected to the size of the liquidity pools - pools with lower liquidity will suffer higher price slippages on large orders.

Assuming current ETH/DAI price is \$2,000 and the initial liquidity pair has 62,500,000 DAI and 25,000 ETH. This will give you a constant product of 1.56 billion. The table below illustrates the price slippage or premium that you will have to pay as your transaction sizes become bigger.

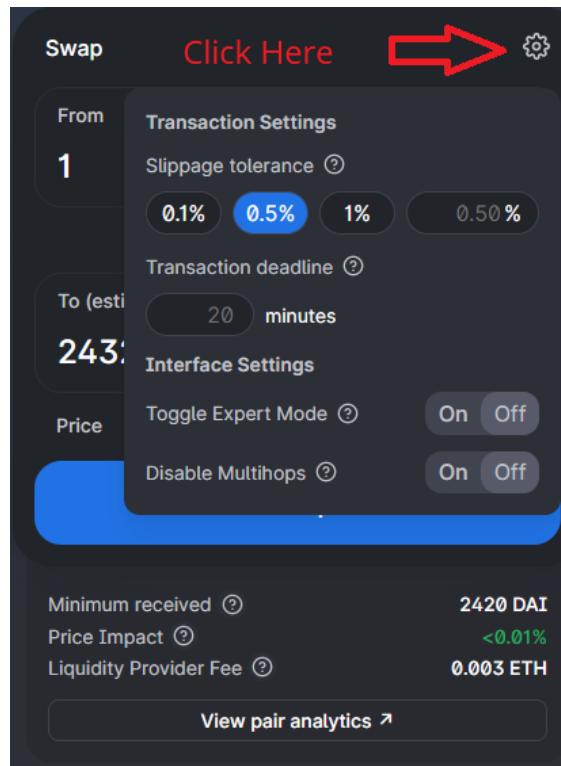
ETH Purchased	Cost per ETH in DAI	Total Cost in DAI	Premium	New DAI Reserve	New ETH Reserve	Product (k)
0	2,500	2,500	0.00%	62,500,000	25,000	1,562,500,000,000
1	2,500.10	2,500	0.04%	62,502,500.10	24,999	1,562,500,000,000
10	2,501.00	25,010	0.04%	62,525,010.00	24,990	1,562,500,000,000
100	2,510.04	251,004	0.40%	62,751,004.02	24,900	1,562,500,000,000
1,000	2,604.17	2,604,167	4.17%	65,104,166.67	24,000	1,562,500,000,000
10,000	4,166.67	41,666,667	66.67%	104,166,666.67	15,000	1,562,500,000,000
20,000	12,500.00	250,000,000	400.00%	312,500,000.00	5,000	1,562,500,000,000
25,000	Infinity	Infinity	Infinity	Infinity	0	1,562,500,000,000

Here is another example of the price slippage example from Uniswap.

Example

Based on ETH/DAI pool on Uniswap, 1 ETH is now worth ~2,433 DAI

You can set your slippage tolerance by going to the transaction setting. Uniswap sets 0.5% as default:



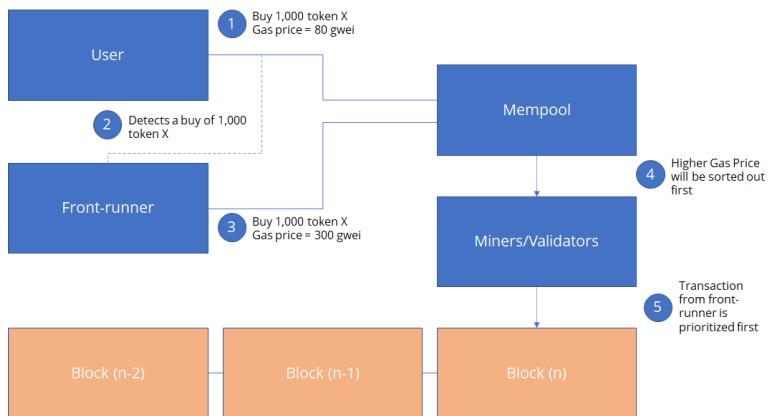
Focusing the indicators on the bottom of the image:

1. **Minimum Received** is the minimum amount of tokens you will receive based on your slippage tolerance. If your slippage tolerance is 0.5%, then the minimum you will receive is $2,433 \text{ DAI} * 0.95\% = 2,420 \text{ DAI}$
2. **Price impact** is the premium that you will have to pay, which will be reflected in the displayed price. The larger your order, the larger the price impact will be.

II. Front-running

As orders made on AMMs are broadcasted to the blockchain for all to see, anyone can monitor the blockchain to pick up suitable orders and front-run it by placing higher transaction fees to have their order mined faster than the target's order. The front-runner who makes this risk-free arbitrage has pulled an attack known as a “Sandwich Attack”.

Below is an illustration of how this could happen:



Below is a snapshot of a “Sandwich Attack” happening with Ampleforth’s governance token (FORTH) on Uniswap.

	Swap Exact Token...	2 mins ago	Uniswap V2: FORTH-US...	out	0x22c1b8c9a9ed05fed1c...	433.42154915555529553	
②	0xd892a918b52c40fbcd...	2 mins ago	Uniswap V2: FORTH-US...	out	0x00000000005736775fb...	648.741261988609786766	
③	0x87113b39133150fa9...	2 mins ago	0x69b9ec0c63d5f2dc05d8...	IN	Uniswap V2: FORTH-US...	159.956132616171413455	
④	Front-runner sell	0x33114e9	2 mins ago	Uniswap V2: FORTH-US...	out	0xc1cc5f78bc1df1f520f1...	
⑤	Victim buy	Swap Exact Token...	2 mins ago	Uniswap V2: FORTH-US...	IN	305.984006609607719949	
⑥	Front-runner buy	0xd149d0c...	2 mins ago	Uniswap V2: FORTH-US...	out	231.342974191964118827	
⑦	0xd1d40b9a9da125e539d...	Delegate	2 mins ago	Uniswap V2: FORTH-US...	out	305.984006609607719949	
⑧	0x05045cf111ee8b46150...	Swap Exact Token...	2 mins ago	0x91d5b0e2cd5cd7018...	IN	129.21179467543208289	
⑨	Front-runner's cost = \$13,088 - \$12,796 - \$61.31 = \$230.69	Profit = \$13,088 - \$12,796 - \$61.31 = \$230.69					
1	Additional Info		FORTH Price = \$41.81 Front runner = buy ~ 306 FORTH for ~ 12,796 USDT Gas price = 126.2 Gwei (\$35.45)				
2	Additional Info		FORTH Price = \$43.29 Victim buy= buy ~ 231 FORTH for ~10,000 USDT Gas price = 126.0 Gwei (\$31.84)				
3	Additional Info		FORTH Price = \$42.77 Front runner = sold ~ 306 FORTH for ~ 13,088 USDT Gas price = 126.0 Gwei (\$25.86)				
①	Swap Exact Token...	2 mins ago	Uniswap V2: FORTH-US...	out	0x22c1b8c9a9ed05fed1c...	433.42154915555529553	
②	0xd892a918b52c40fbcd...	2 mins ago	Uniswap V2: FORTH-US...	out	0x00000000005736775fb...	648.741261988609786766	
③	0x87113b39133150fa9...	2 mins ago	0x69b9ec0c63d5f2dc05d8...	IN	Uniswap V2: FORTH-US...	159.956132616171413455	
④	Front-runner sell	0x33114e9	2 mins ago	Uniswap V2: FORTH-US...	out	0xc1cc5f78bc1df1f520f1...	
⑤	Victim buy	Swap Exact Token...	2 mins ago	Uniswap V2: FORTH-US...	IN	305.984006609607719949	
⑥	Front-runner buy	0xd149d0c...	2 mins ago	Uniswap V2: FORTH-US...	out	231.342974191964118827	
⑦	0xd1d40b9a9da125e539d...	Delegate	2 mins ago	Uniswap V2: FORTH-US...	out	305.984006609607719949	
⑧	0x05045cf111ee8b46150...	Swap Exact Token...	2 mins ago	0x91d5b0e2cd5cd7018...	IN	129.21179467543208289	
⑨	Front-runner's cost = total gas = \$61.31	Profit = \$13,088 - \$12,796 - \$61.31 = \$230.69					

III. Impermanent Loss

Another downside of AMM is the impermanent loss that happens when you provide liquidity to the AMMs. Impermanent loss is similar to measuring your opportunity cost of holding the token within the pools versus holding them in your wallet. Note: the loss is not realized until you remove your tokens from the liquidity pool.

The higher the divergence between the value of holding your tokens in the pool and wallet, the higher is the impermanent loss.

Example

Assuming you created an ETH/DAI pool on Uniswap by providing 10,000 DAI and 5 ETH to the pool.

- Price of 1 ETH = 2,000 DAI
- The pool consists of 5 ETH and 10,000 DAI
- Pool liquidity uses the Constant Product Market Maker formula”
 $(x * y = k) \rightarrow 5 * 10,000 = 50,000$

Say the price of ETH doubles to 4,000 DAI.

- Arbitrageurs will arbitrage the difference in the ETH price quoted in Uniswap until it reaches 1 ETH = 4,000 DAI
- The pool will rebalance the reserve ratio until it matches the pool constant at 50,000
- The new pool ratio would become 3.536 ETH and 14,142 DAI

To calculate your impermanent loss, you can subtract your gains from holding outside the pool and within the pool.

At 1 ETH = 2,000 DAI, your original capital (5 ETH and 10,000 DAI) is valued at 20,000 DAI.

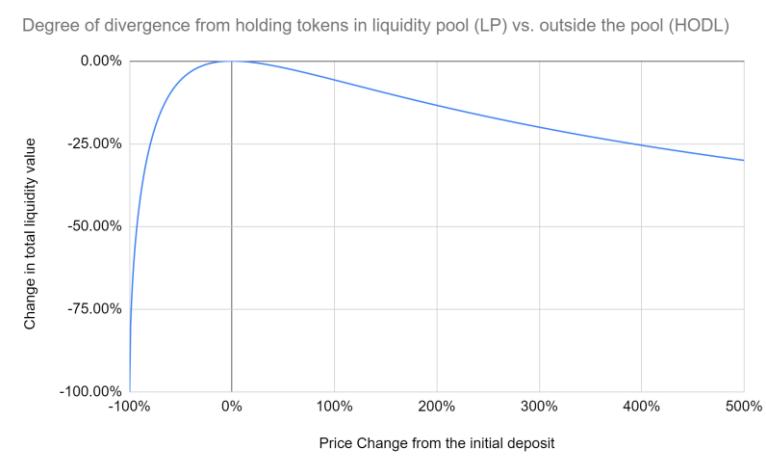
At 1 ETH = 4,000 DAI

- Your holding within the pool would be:

- $(3.536 \text{ ETH} * 4,000 \text{ DAI}) + 14,142 \text{ DAI} = 28,286 \text{ DAI}$
- Return on investment = +41% (ignoring earnings from trading fee)
- Your holding outside the pool would be:
 - $(5 \text{ ETH} * 4,000 \text{ DAI}) + 10,000 \text{ DAI} = 30,000 \text{ DAI}$
 - Return on investment = +50%
- Your impermanent loss is thus:
 - $30,000 \text{ DAI} - 28,286 \text{ DAI} = 1,716 \text{ DAI}$

This loss is only realized if you withdraw your liquidity from Uniswap.

The graph below shows the opportunity cost you will get if you hold your tokens outside the pool and within the pool.



- 1.25x price change = 0.6% loss relative to HODL
- 1.50x price change = 2.0% loss relative to HODL
- 1.75x price change = 3.8% loss relative to HODL
- 2x price change = 5.7% loss relative to HODL

- 3x price change = 13.4% loss relative to HODL
- 4x price change = 20.0% loss relative to HODL
- 5x price change = 25.5% loss relative to HODL

Thus, trading pairs that trade within a small price range (e.g. stablecoins) are less exposed to impermanent loss.

Notable Mentions

-  **PancakeSwap**
PancakeSwap is a fork of Uniswap, but it is built on top of the Binance Smart Chain blockchain. It is the biggest AMM on Binace Smart Chain, and its volume is even higher than Uniswap as of April 2021.
-  **TerraSwap**
TerraSwap exists on Terra Chain, and it is the only DEX protocol on the Terra blockchain. You can choose your trading fees to be denominated in any Terra Chain assets.
-  **0x Protocol**
0x is a DEX infrastructure layer with two main offerings: a DEX aggregation API that allows projects to launch whitelabel DEX and a consumer-facing DEX aggregator called Matcha. Projects such as Tokenlon, Metamask, Zapper, and Zerion have integrated 0x to launch their exchange services.

Conclusion

DEXs play a vital role in powering the DeFi space as it depicts the current market behavior, especially the price and liquidity of cryptocurrencies. It determines the value of various cryptocurrencies relative to each other and illustrates the dynamic nature of trades and capital flow.

Recommended Readings

1. Understanding AMM Basics
<https://defiweekly.substack.com/p/understanding-amms-the-basics-f30>
2. Types of AMMs
<https://blog.chain.link/challenges-in-defi-how-to-bring-more-capital-and-less-risk-to-automated-market-maker-dexs/>
3. Understanding Price Impact on AMM
<https://research.paradigm.xyz/amm-price-impact>
4. Front-running Issue on Ethereum
<https://www.coindesk.com/new-research-sheds-light-front-running-bots-ethereum-dark-forest>
5. Uniswap V3
<https://uniswap.org/blog/uniswap-v3/>

CHAPTER 4: DEX AGGREGATORS

Liquidity is essential to ensure trades can be executed without severely affecting the market price. The DEX market is extremely competitive, with multiple DEXs competing for users and liquidity. Liquidity is thus often disparate and leads to inefficient capital management.

While the impact on smaller transactions may be inconsequential, larger DEX transactions will be prone to higher price slippage. This is where DEX aggregators help traders with the best price execution across the various DEXs.

DEX aggregators look for the most cost-effective transactional routes by pooling liquidity from different DEXs. By routing a single transaction across multiple liquidity pools, traders making large trades can take advantage of gas savings and minimize the cost of price impacts due to low liquidity.

We covered DEX aggregators briefly in our *How to DeFi: Beginner* book using 1inch as an example. In the next section we will cover newer DEX aggregators such as Matcha and Paraswap, and offer some comparisons between each protocol.

DEX Aggregator Protocols

1inch Network



1inch Network is a DEX aggregator solution that searches for cheaper rates across multiple liquidity sources. The initial protocol incorporates the Pathfinder algorithm which looks for optimal paths among different markets. Since its inception on the Ethereum network, 1inch has expanded to support the Binance Smart Chain and Polygon networks. The 1inch Aggregation Protocol has also undergone two significant updates, and has been on version 3 since March 2021.⁸

As of 31 May 2021, there are 50+ liquidity sources on Ethereum, 20+ liquidity sources on Binance Smart Chain, and 10+ liquidity sources on Polygon. Notably, in just two years, the 1inch DEX aggregator has surpassed \$40B in overall volume on the Ethereum network alone.

Unlike other DEX aggregators, 1inch has two native tokens. One is a gas token (CHI), and the other is a governance token (1INCH).

CHI is a gas token that takes advantage of the Ethereum storage refund. Gas tokens help smart contracts erase unnecessary storage during the transaction process and reduce gas fees. You can think of CHI as a discount coupon that can be redeemed for cheaper transactions, allowing users to save up to 42% of their gas fees.

⁸ 1inch Network, (2021, March 16). *Introducing the 1inch Aggregation Protocol v3*. Medium. <https://blog.1inch.io/introducing-the-1inch-aggregation-protocol-v3-b02890986547>.

The 1INCH token, released in December 2020, propelled the protocol into becoming a more decentralized entity. Holders of 1INCH allow the community to vote for specific protocol settings under the Decentralized Autonomous Organization (DAO) model. The governance model enables stakers to control two main aspects:⁹

- 1) **Pool governance** - governs specific parameters for each pool, such as the swap fee, the price impact fee, and the decay period.
- 2) **Factory governance** - governs general parameters for all pools, such as the default swap fee, the default price impact fee, the default decay period, the referral reward, and the governance reward.

Also, through staking of the 1INCH token, users earn the Spread Surplus (positive slippage), which is the net positive difference between swap transactions when the executed price is slightly better than the price quoted. Notably, the 1inch Network also has many partnerships with other protocols where liquidity mining incentives are commonplace for 1INCH trading pairs.

Other notable features include limit orders and the option to select Pathfinder's routing process and choose between receiving maximum returns or minimizing gas costs.

The 1inch Network also incorporates its own Liquidity Protocol. The automated market maker protects users from front-running attacks and offers more opportunities for liquidity providers.

⁹ 1inch Network. (2020, December 25). *1INCH token is released*. Medium.
<https://medium.com/1inch-network/1inch-token-is-released-e69ad69cf3ee>

Matcha



Matcha

Matcha is a decentralized exchange (DEX) aggregator built by 0x Labs. Matcha is powered by 0x protocol, a protocol with various products, including a peer-to-peer network for sharing orders (0x Native Liquidity) and their proprietary API.¹⁰ Matcha pulls data from the 0x API and efficiently routes orders across all the available liquidity sources (20+ as of 31 May 2021).

Unlike other DEX aggregators, Matcha utilizes a combination of on-chain and off-chain components throughout the trading experience. Quotes are generated off-chain via the 0x API to minimize gas costs before being utilized on-chain to execute orders.¹¹ The 0x API finds the most cost-effective trading path (including gas costs) and can even split individual orders across multiple liquidity sources automatically if it's better for the trader to do so.

To date, there have been four major updates to 0x's API (which powers Matcha), with the latest being 0x version 4 released in March 2021. Through this version 4 update, Matcha users should expect more gas-efficient orders (up to 70% gas saved for quote orders and 10% for limit orders) and better overall prices.¹²

Since 0x version 3, Matcha users are charged a small protocol fee (paid in ETH) on 0x open order book liquidity. The fee is proportional to the gas cost of filling orders and scales linearly with gas price. What is important to

¹⁰ Kalani, C. (2020, June 30). *Say hello to Matcha!* Matcha. <https://matcha.xyz/blog/say-hello-to-matcha>.

¹¹ Brent. (n.d.). *Does Matcha have a trading API?* Matcha. <http://help.matcha.xyz/en/articles/3956594-does-matcha-have-a-trading-api>.

¹² Gonella, T. (2021, January 27). *Say hello to 0x v4*. Medium. <https://blog.0xproject.com/say-hello-to-0x-v4-ce87ca38e3ac>.

note here is that Matcha technically does not charge transactions other than the requisite network fees required by either the native chain or liquidity source.

Unlike 1inch, Matcha shares all positive slippage back to the user. Other notable features include limit orders and Matcha's recent support of the Binance Smart Chain network and Polygon network.

Paraswap



ParaSwap was first developed in September 2019 and uses its own routing algorithm, Hopper. ParaSwap examines the rate for the given pair on all supported exchanges and displays the effective rate (accounting for slippage) for each pair.

ParaSwap implements several solutions to reduce gas usage across the platform, such as implementing the REDUX gas token.¹³ Gas costs are taken into account when analyzing swapping paths.

ParaSwap's most recent update, version 3, was introduced in January 2021. It included a significant UI upgrade and improved swapping contracts.¹⁴ Emphasis was placed on reducing overall gas costs by 30%, especially for trades settled using only one DEX.

Protocol revenue is generated through two main avenues.¹⁵ The first is through third-party integrators, where if they charge a fee on facilitated

¹³ Paraswap. (2021, April 11). *What is REDUX gas token?* Medium. <https://paraswap.medium.com/what-is-redux-gas-token-cc20cc55fbd7>.

¹⁴ Paraswap. (2021, January 28). *Introducing ParaSwap's new UI & a significant upgrade for our contracts.* Medium. <https://medium.com/paraswap/introducing-paraswaps-new-ui-a-significant-upgrade-for-our-contracts-ed15d632e1d0>.

¹⁵ *Fee Structure.* Learn2Swap. (n.d.). <https://doc.paraswap.network/understanding-paraswap/fees>.

swaps, ParaSwap takes a 15% portion of the fee. The second is through positive slippage, where 50% is directed to the protocol and the other 50% is shared back with the user.

ParaSwap currently has 48 sources of liquidity. This is supplemented by native pools (ParaSwapPools), which are supplied by private market makers. ParaSwap also recently integrated with the Binance Smart Chain network and Polygon network.

DEX Aggregator's Performance Factors

There are many intricacies involved under the hood of DEX aggregators, making it difficult to compare them fairly. While users might focus on the quoted price, they are not necessarily reliable. Here's why:

Example

Let's say a user wants to swap 1,000 USDC for 1,000 USDT.

Aggregator X quotes 1,000 USDT and has an estimated transaction cost of 5 USDT, giving a realized exchange rate of $1 \text{ USDC} = 0.995 \text{ USDT}$. After swapping 1,000 USDC, the user will receive 995 USDT.

Aggregator Y quotes 1,005 USDT and has an estimated transaction cost of 15 USDT, giving a realized exchange rate of $1 \text{ USDC} = 0.990 \text{ USDT}$. After swapping 1,000 USDC, the user will receive 990 USDT.

In this example, Aggregator X is more cost-efficient after taking into consideration the transaction fee. You have to remember that this example uses estimated figures that the DEX aggregator provides before swapping.

In reality, when a person performs a swap, the time difference between approving the exchange and the successful execution of the swap on-chain will affect the final price. During that period, external market forces such as network congestion and the size of selected liquidity pools may change. The

protocol's routing algorithm will also affect the outcome as more efficient transactions reduce network usage and minimizes failed transactions.

Another point is the size of the transaction. The cost savings incurred from DEX aggregators are proportionally higher for larger transactions because they are more prone to higher price slippage. Smaller transactions may not need to rely on different liquidity pools because a single liquidity pool is the most optimal route.

If we categorize all these metrics, we get four primary factors that determine a DEX aggregator's performance:

1. Routing Algorithm
2. Sources of Liquidity
3. Current Market State
4. Size of Transaction

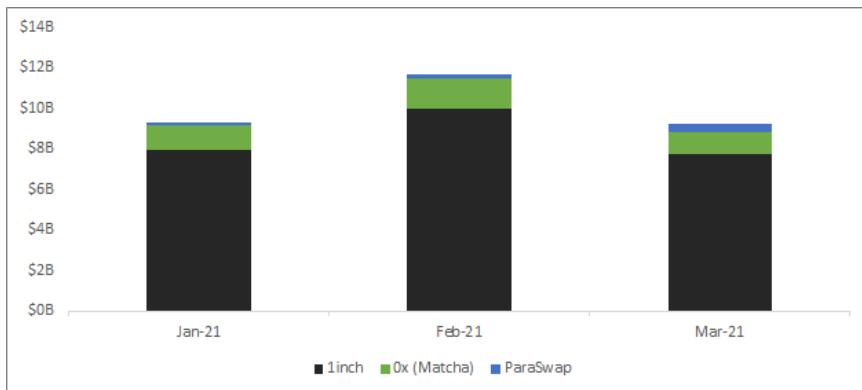
Which DEX Aggregator offers the most value?

DEX aggregators have become an essential part of the DEX economy. While it is difficult to ascertain which DEX aggregator offers the most value, the following table does offer some clarity:

As of 31 May 2021	1inch	Matcha	ParaSwap
Sources of Liquidity	80+	20+	48
Token	1INCH + CHI	None	None
Routing Algorithm	Pathfinder	0x API	Hopper
Limit Orders	Yes	Yes	No
Staking Features	Yes	No	No

As of 31 May 2021	1inch	Matcha	ParaSwap
Protocol Fees	Variable. At time of writing is 0.25% fixed fee on all trades.	70,000 gwei * Gas price of the transaction (where applicable)	None for users but third-party integrators are charged a 15% fee on facilitated swaps
Blockchains Supported	Ethereum and Binance Smart Chain, and Polygon	Ethereum, Binance Smart Chain, and Polygon	Ethereum, Binance Smart Chain, and Polygon
Transfer of Positive Slippage to User	Variable - At time of writing (May 2021), about 20% of positive slippage is given to referrers and 80% for 1INCH stakers.	100%	50%

1inch has a lot of first-mover advantages. As of 31 May 2021, the protocol has the most sources of liquidity, with over 80+ sources. 1inch is also the only DEX aggregator with its own native tokens, giving it a distinct advantage over other protocols and allowing users to stake 1INCH tokens and earn protocol fees. 1inch is also more decentralized than other protocols which lack a DAO. All these advantageous are reflected in trading volume, the most basic metric:



Source: Dune Analytics

The total trading volume for Q1 2021 is dominated by 1inch. In March 2021, 1inch had 84.2% of the total market share and \$7.76 billion worth of trading volume. Of course, this could also be caused by a variety of reasons, including user loyalty and information asymmetry. However, when taken a whole, high-user retention rates suggest that the market recognizes 1inch's benefits.

Associated Risks

It is good practice to not treat quoted prices on DEX aggregators as gospel. While DEX aggregators aim to ensure that the executed transaction conforms to the quoted price, this does not always occur.

Another point is the size of the transaction. Although DEX aggregators offer better cost savings for larger transactions, it may sometimes be better for smaller traders to interact directly with a DEX.

DEX aggregators are usually reliable, but there have been instances where transactions are routed through small and illiquid pools. As a user, you should always check that your slippage is not too high before approving a transaction.

Notable Mentions



- **DEX.AG (rebranded to Slingshot)**

DEX.AG is one of the smaller DEX aggregators which uses its own proprietary routing algorithm, X Blaster. The project rebranded itself to Slingshot in November 2020. At the time of writing (1 April 2021), the protocol is integrated with 18 liquidity sources, does not take any trading fees and has yet to release a live version of their update.



- **Totle**

Totle is another small DEX aggregator which relies on their native API (Totle API). At the time of writing (1 April 2021), there are 15 liquidity sources.

Conclusion

DEXs are the lifeblood of DeFi. However, for many power users (whales especially), DEX aggregators are even more important as DEX aggregators can offer better cost efficiency for large transactions. DEX aggregators have even evolved to a point where they have their own liquidity pools, further blurring the line between DEX aggregators and DEXs.

The DEX sector is a prime example of DeFi composability. DEX aggregators are built on top of DEXs, to serve different user profiles. Thus, we benefit from a more comprehensive suite of innovative products borne out of increased competition and mutualistic integration.

Recommended Readings

1. Overview of DEX Aggregators
<https://www.delphidigital.io/reports/defi-aggregators/>
2. Comparing Different DEX Aggregators
<https://medium.com/2key/defi-dexes-dex-aggregators-amms-and-built-in-dex-marketplaces-which-is-which-and-which-is-best-fba04ca48534>
3. 0x's October 2020 Study on Dex Aggregators
<https://blog.0xproject.com/a-comprehensive-analysis-on-dex-liquidity-aggregators-performance-dfb9654b0723>
4. 1inch's v3 Upgrade and Comparison with Other Dex Aggregators
<https://blog.1inch.io/introducing-the-1inch-aggregation-protocol-v3-b02890986547>

CHAPTER 5: DECENTRALIZED LENDING & BORROWING

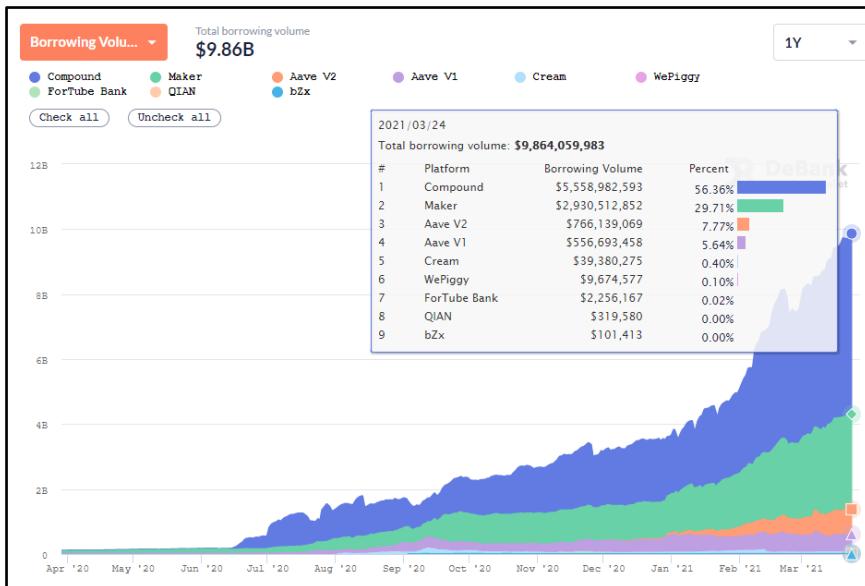
The capital market in the traditional financial system is not accessible by many - only the rich have the VIP card to access it.

Imagine you are a venture capitalist looking to finance your next business venture. You can take a loan and offer your assets as collateral. The collateralized capital will remain untouched and continue to grow over time, and can be redeemed at a later date. Of course, this is not a risk-free strategy. If you are not careful, you may default on your loan payment and lose your collateral.

Lending protocols in DeFi have now democratized access to debt for everyone. Using your cryptocurrencies as collateral, you can borrow from these protocols and leverage upon it. As one of the largest DeFi categories, decentralized lending and borrowing has grown exponentially, with borrowing volume reaching \$9.7 billion in April 2021. That is an increase of 102 times compared to the year before!

The leading DeFi lending protocols are Compound, Maker, Aave, and Cream.

Decentralized Lending and Borrowing



Source: DeBank

Overview of Lending & Borrowing Protocols

Compound



Compound Finance is a money market protocol built by Compound Labs. It is an Ethereum-based, open-source money market protocol where anyone can lend or borrow cryptocurrencies frictionlessly. As of 1 April 2021, there are nine different tokens available on the Compound Platform.

1. 0x (ZRX)
2. Basic Attention Token (BAT)
3. Compound (COMP)
4. Dai (DAI)

5. Ether (ETH)
6. USD Coin (USDC)
7. Tether (USDT)
8. Uniswap (UNI)
9. Wrapped Bitcoin (WBTC)

Compound operates as a liquidity pool built on the Ethereum blockchain. Suppliers supply assets to the liquidity pool to earn interest, while borrowers take a loan from the liquidity pool and pay interest on their debt. In essence, Compound bridges the gaps between lenders who wish to accrue interest from idle funds and borrowers who want to borrow funds for productive or investment use.

In Compound, interest rates are denoted in Annual Percentage Yield (APY), and the interest rates differ between assets. Compound derives the interest rates via algorithms that take into account the supply and demand of the assets.

Essentially, Compound lowers the friction for lending/borrowing by allowing suppliers/borrowers to interact directly with the protocol for interest rates without needing to negotiate loan terms (e.g., maturity, interest rate, counterparty, collaterals), thereby creating a more efficient money market.

Compound is the largest DeFi lending platform by borrowing volume, with a 56% market share (Debank, 1 April 2021). In June 2020, Compound introduced its governance token, Compound (COMP).

Maker



Maker is the oldest DeFi borrowing protocol. It enables over-collateralized loans by locking more than 30 tokens supported in a smart contract to mint DAI, a decentralized stablecoin pegged to the USD.¹⁶ Besides being a borrowing protocol, Maker also acts as a stablecoin issuer (DAI).

On 19 December 2017, Maker originally started with the Single Collateral DAI (SAI). It was minted using Ether (ETH) as the sole collateral. On 18 November 2019, Maker upgraded SAI to Multi-Collateral DAI (DAI), which can be minted with 29 different tokens as collateral.

Maker now even accepts USDC, a centralized stablecoin to help manage DAI price instability. Maker has made huge progress in bridging the gap with traditional finance by onboarding the first real-world asset as collateral through Centrifuge. On 21 April 2021, the company successfully executed its first MakerDAO loan for \$181k with a house as collateral, effectively creating one of the first blockchain-based mortgages.

Unlike other lending protocols, users cannot lend assets to Maker. They can only borrow DAI by depositing collateral. DAI is the biggest decentralized stablecoin and has seen growing adoption in the DeFi ecosystem. We will look deeper into DAI in [Chapter 6](#).

¹⁶ (2021, May 10). Introducing The Redesigned Oasis Borrow - Oasis Blog - Oasis.app. Retrieved May 27, 2021, from <https://blog.oasis.app/introducing-the-redesigned-oasis-borrow/>

Aave



Aave is another prominent decentralized money market protocol similar to Compound. As of April 2021, users can lend and borrow 24 different assets on Aave, significantly more compared to Compound.

Both Compound and Aave operate similarly where lenders can provide liquidity by depositing cryptocurrencies into the available lending pools and earn interest. Borrowers can take loans by tapping into these liquidity pools and pay interest.

Aave distinguished itself from Compound by pioneering new lending primitives like rate switching, collateral swap, and flash loans.

Rate switching: Borrowers on Aave can switch between variable and stable interest rates.

Collateral Swap: Borrowers can swap their collateral for another asset. This helps to prevent loans from going below the minimum collateral ratio and face liquidation.

Flash loans: Borrowers can take up loans with zero collateral if the borrower repays the loan and any additional interest and fees within the same transaction. Flash loans are useful for arbitrage traders as they are capital-efficient in making arbitrage trades across the various DeFi Dapps.

Cream Finance



Cream Finance (C.R.E.A.M) was founded in July 2020 by Jeffrey Huang and Leo Cheng. Cream is a Compound fork that also services long-tail, exotic DeFi assets. Cream partnered (merged) into the Yearn Finance ecosystem in November 2020.¹⁷ It is deployed across Ethereum, Binance Smart Chain, and Fantom.

Cream has a more lenient asset onboarding strategy as compared to Compound and Aave. Employing the fast-mover strategy, it has listed more assets than any other lending protocol at a faster speed. It has chosen to focus on long-tail assets - assets with lower liquidity or belong in niche categories. It was one of the first lending protocols to accept yield-bearing tokens and LP tokens as collateral.

Cream has also launched the Iron Bank, an uncollateralized lending service offered to whitelisted partners. As one of its partners, Yearn Finance can utilize the borrowed funds from Cream to further increase the yield obtained from its yield farming activities.

In anticipation of the upcoming launch of ETH 2.0, Cream also offers ETH 2.0 staking service where users can stake ETH for CRETH2, which can be supplied and borrowed against as collateral. All the ETH 2.0 staking work will be done by Cream, and the yield is shared with CRETH2 holders. Essentially, CRETH2 is a custodial staking service, with a fee of 8% on the validator reward. In addition to ETH 2.0, Cream also offers staking services for Binance Smart Chain and Fantom.

¹⁷ (2020, November 25). Yearn & Cream v2 merger. Yearn and Cream developers ... - Medium. Retrieved May 28, 2021, from <https://medium.com/iearn/yearn-cream-v2-merger-e9fa6c6989b4>

Cream positions itself as a more risky lender against Compound and Aave, facilitating and enabling the market demand for leveraging and shorting niche assets.

Protocols Deep Dive

(US\$ in millions) As of 24th March 2021	Compound	Maker	Aave	Cream
Number of Assets Supported				
Collateral Type	8	29	21	47
Borrowing Type	9	1	25	65
Borrowing Volume	\$ 5,559	\$ 2,931	\$ 1,324	\$ 39
Total Value Locked (TVL)	\$ 6,910	\$ 6,090	\$ 5,010	\$ 202
Utilization Ratio (Borrowing Vol/TVL)	0.80	0.48	0.26	0.19

Source: Compound, Maker, Aave, Cream, DeBank, Token Terminal

The table above gives information about the 4 largest lending protocols: Compound, Maker, Aave, and Cream.

We will go through each metric in sequence to assess the protocol's capital efficiency in terms of borrowing volume and total value locked (TVL). Subsequently, we will look at the associated risks for each of them.

Assets Supported

To ensure trustless loans can happen, borrowers will need to deposit assets (collateral) with greater value than the amount borrowed. This is known as over-collateralization and underpins the solvency of DeFi lending protocols. How much borrowers can borrow depends on the collateral ratio of each asset on the various DeFi lending protocols.

Amongst the DeFi lending protocols, Cream has the most supported assets - 45 assets can be used as collateral, and 65 assets available for borrowing.

In contrast, Compound has the least number of supported assets - only eight assets can be used as collateral and nine assets available for borrowing. Compound is more conservative with its asset onboarding strategy.

Revenue

One of the vital metrics for lending platforms is their borrowing volume. This metric is important because borrowers pay fees for their loans and thereby produce revenue for these protocols. Here is the breakdown of how each protocol earns revenues:

- **Compound:** A portion of the interest paid by the borrower will go to its reserve, which acts as insurance and is controlled by COMP token holders. Each supported asset has a reserve factor that will determine how much goes into the reserve.¹⁸ You can check each of the reserve factors by clicking on the respective asset pages.
- **Maker:** When borrowers repay their loans, they will pay the principal along with the interest fee that is determined by the stability fee. Each supported collateral has its stability fee.¹⁹
- **Aave:** The platform has 2 types of fees:²⁰
 - 0.00001% of the loan amount is collected on loan origination in Aave V1.
 - 0.09% is collected from the flash loan amount—more about flash loans in [Chapter 14](#).
- **Cream:** A portion of the interest paid from the borrowers to suppliers goes to Cream's Reserve Protocol and is distributed to CREAM token holders as rewards.²¹

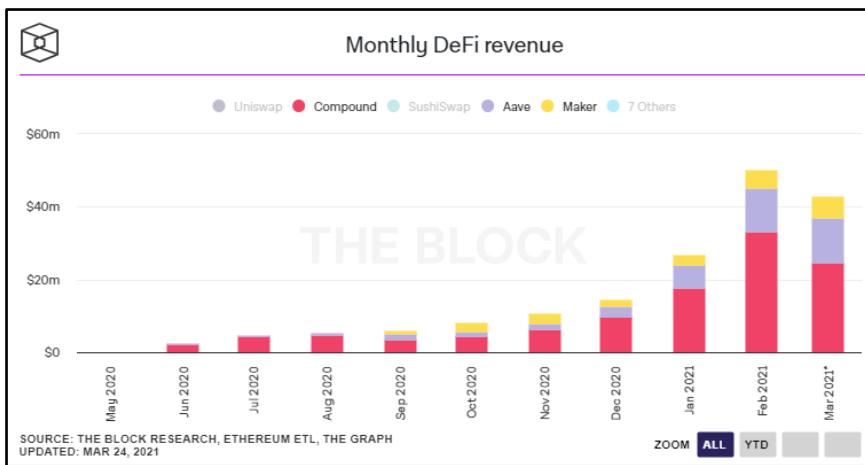
As of April 2021, Compound generates the highest revenue among the lending protocols. See below:

¹⁸ (2018, December 5). Compound FAQ - Medium. Retrieved March 25, 2021, from <https://medium.com/compound-finance/faq-1a2636713b69>

¹⁹ (n.d.). Oasis Borrow - Oasis.app. Retrieved May 22, 2021, from <https://oasis.app/borrow/markets>

²⁰ (n.d.). Introduction to Aave - FAQ - Aave Document Portal. Retrieved March 25, 2021, from <https://docs.aave.com/faq/>

²¹ (2020, September 3). CREAM Reserve Protocol. The Reserve Protocol is the fee ... - Medium. Retrieved March 25, 2021, from <https://medium.com/cream-finance/c-r-e-a-m-reserve-protocol-99f811e693e4>

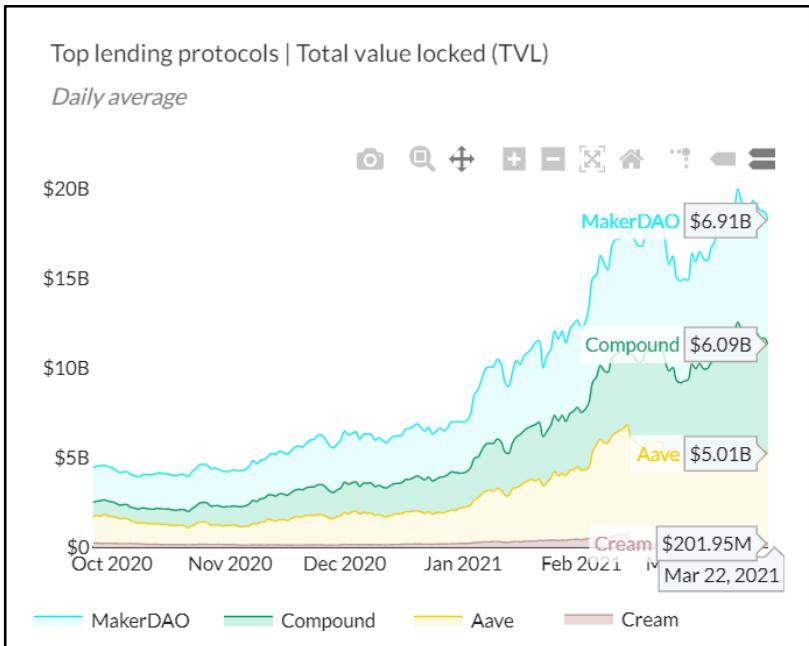


Source: TheBlockResearch Data Dashboard

Total Value Locked (TVL)

While all of the lending protocols have seen their TVL grow since October 2020, it is worth noting that the TVL for both Aave and Maker are more organic than Compound. This is because Compound has ongoing incentives for borrowers and lenders in their liquidity mining program.

Note: Aave recently launched a liquidity mining program at the end of April 2021.



Source: TokenTerminal

Utilization ratio (Borrowing Volume/TVL)

DeFi lending and borrowing protocols derive protocol revenue from their borrowings. Meanwhile, TVL in lending platforms is driven by users depositing their assets to earn a yield or to be used as borrowing collateral.

By dividing the above two figures, we can derive the utilization ratio of each protocol - the higher the utilization ratio, the more efficient the TVL is being put to work.

(US\$ in millions) As of 24th March 2021	Compound	Aave	Cream
Number of Assets Supported			
Collateral Type	8	21	47
Borrowing Type	9	25	65
Borrowing Volume	\$ 5,559	\$ 1,324	\$ 39
Total Value Locked (TVL)	\$ 6,910	\$ 5,010	\$ 202
Utilization Ratio (Borrowing Vol/TVL)	0.80	0.26	0.19

*Maker is not included in this table because unlike its peers, the collateral cannot be borrowed.

At a glance, Compound seems like it has the highest utilization ratio at 0.80, and Cream the lowest at 0.19. However, Compound's high utilization rate

may be due to its liquidity mining program that rewards borrowers with COMP tokens, while Cream's low ratio may be because it has suffered a couple of exploits in the past.^{22,23} Given security concerns, Cream has stepped up their security coverage with a \$1.5 million bug bounty with Immunefi, insurance coverage with Armor.fi/Nexus Mutual, and transparency report with Defi Safety.²⁴

Lending and Borrowing Rates

Now, let us discuss the two primary users of lending protocols: the lenders and the borrowers.

Lenders

Supply Market (Snapshot on 24th March 2021)

	Compound	Aave	Cream
USDC	5.51%	7.50%	16.13%
ETH	0.16%	0.33%	1.60%
DAI	7.75%	8.71%	13.50%
WBTC	0.45%	0.18%	2.22%
USDT	2.43%	7.13%	26.11%

Source: Compound, Aave, Cream

Based on the table above, Cream provides the highest APY and Compound provides the lowest APY for most assets. As a lender, although it may seem like it is most desirable to earn the highest APY on Cream, the caveat with high APYs often means higher underlying risks of depositing your capital in these lending protocols relative to its peers.

²² (2021, March 15). Another DeFi Hack: PancakeSwap, Cream Finance ... - Crypto News. Retrieved April 28, 2021, from <https://cryptonews.com/news/another-defi-hack-PancakeSwap-cream-finance-websites-comprom-9548.htm>

²³ (2021, February 13). DeFi Protocols Cream Finance, Alpha Exploited in Flash ... - CoinDesk. Retrieved April 28, 2021, from <https://www.coindesk.com/defi-protocols-cream-finance-alpha-lose-37-5m-in-exploit-prime-suspect-idd>

²⁴ (2021, April 20). CREAM Finance Is Working With Immunefi, Armor.fi, and ... - Medium. Retrieved May 28, 2021, from <https://medium.com/cream-finance/security-immunefi-armorfi-defisafety-aa6e9e7c50e8>

Cream has to offer high APYs to attract capital to its protocol as it has been hacked several times previously. Overall, Compound provides the lowest APY for all supported assets relative to its peers as its brand entity focuses on the platform's security.

Borrowers

Borrow Market (Snapshot on 24th March 2021)

	Compound	Aave (Variable rate)	Aave (Stable rate)	Cream	Maker
USDC	7.25%	8.23%	15.23%	23.20%	-
ETH	2.76%	2.06%	5.57%	8.02%	-
DAI	10.78%	14.71%	22.71%	21.26%	0% - 9% *
WBTC	4.82%	1.61%	5.01%	9.80%	-
USDT	3.67%	6.80%	14.80%	42.04%	-

* The interest depends on the collateral assets deposited on Maker

Source: Compound, Aave, Cream, Maker

As for borrowers, you would generally look at lending platforms that offer the lowest interest rate because that means cheaper borrowing costs. Compound offers the most competitive borrowing rates relative to its peers. However, Compound has a limited list of assets that users can borrow.

As for Aave, it provides two rate options for borrowers: variable rate and stable rate. Users can switch their rates at any point of time, whichever is cheaper.

Cream charges the highest borrowing rates - primarily because it has the most diverse list of assets for users to borrow. The higher borrowing rates on Cream are needed to compensate lenders' high APY requirement for taking the additional risk of supplying their capital.

Interestingly enough, some of Cream's borrowers are institutions using Cream's Iron Bank, such as Alpha Finance and Yearn Finance. Iron Bank provides a novel uncollateralized lending service where other protocols do not have to provide collateral to borrow from Cream.

Now, let's focus specifically on DAI's borrowers. Maker is the sole issuer of DAI, and it provides the cheapest interest rate to any of its lending peers using any of the supported assets. Maker's borrowing volumes are the

second-largest behind Compound, and are locked mainly to mint DAI, which as of 1 April 2021 is the fourth-largest stablecoin in the world. This makes Maker an incredibly important player in the DeFi ecosystem.

Associated Risks

When using decentralized lending and borrowing protocols, you have to be aware of technical risks such as smart contract bugs. Hackers may exploit and drain the collateral held on these protocols if the developers are not careful with their code deployment.

Additionally, many lending protocols rely on price oracles (more on this in [Chapter 12](#)) to provide on-chain price data. Price oracles may fail or can be exploited. In November 2020, Compound's price oracle was exploited by driving up the price of DAI by 30% on Coinbase Pro, resulting in \$89 million worth of loans being liquidated.²⁵

However, as borrowers, the main risks involve losing your collateral through mismanagement of your collateral ratio. Cryptocurrency is known for its extreme price volatility, and you are at risk of having your loan go below its minimum collateral ratio and having your collateral liquidated. This is not an ideal outcome as you will take a considerable loss and pay a liquidation fee. It is thus vital to constantly monitor and maintain a healthy collateral ratio for your loans.

²⁵ (2020, November 26). Oracle Exploit Sees \$89 Million Liquidated on Compound - Decrypt. Retrieved May 14, 2021, from <https://decrypt.co/49657/oracle-exploit-sees-100-million-liquidated-on-compound>

Notable Mentions

-  **Venus**
Venus Protocol is a money market and stablecoin protocol that operates on Binance Smart Chain. The protocol, initially incubated by Binance, is a fork of Compound and MarkerDAO. Swipe took over the work of developing the Venus Protocol since acquired by Binance.
-  **Anchor**
Anchor Protocol is a lending protocol operating on the Terra blockchain. The lending mechanics on the platform work similarly to Compound and Aave. Anchor Protocol has a 20% yield target for UST, a USD stablecoin native on the Terra blockchain.
-  **Alchemix**
Alchemix differentiates itself from other lending players by introducing self-payable loans without the risk of getting liquidated. Simply put, your collateral will be used to earn interest, and the interest earned will be used to repay the loan you made on Alchemix.
-  **Liquity**
Liquity is a DeFi borrowing protocol that lets you draw its stablecoin, Liquity USD (LUSD), without an interest fee. You will have to use Ethereum as collateral and maintain a minimum collateral ratio of only 110%. The repayment made will be in LUSD. The loans are secured by a Stability Pool containing LUSD and by fellow borrowers collectively acting as guarantors of last resort.

Conclusion

DeFi lending and borrowing protocols have seen incredible adoption - they have been dominating DeFi TVL ranking charts. However, the majority of DeFi lendings are currently still overcollateralized, signifying poor capital efficiency.

Traditional lenders utilize credit scores derived from personal information such as job, salary, and borrowing history before deciding to grant a loan. In DeFi, it is tough to develop a credit history with pseudonymous identities.

Several protocols aim to make progress on undercollateralized loans such as TrueFi, Cream, and Aave. In these protocols, selected whitelisted entities can obtain a loan without posting any collateral.

Undercollateralized loans will be the next stage of development for DeFi lending and borrowing protocols. Undercollateralized loans will be needed so that DeFi lending and borrowing protocols can be more capital efficient and compete effectively against traditional lenders.

While DeFi lending is currently still mainly constrained within the digital asset universe, with few links to real-life assets, Maker has made huge progress in April 2021 by accepting real-world assets such as real estate as collateral.²⁶ Thus, one may presume better integration between Traditional Finance and DeFi in the future.

²⁶ “DeFi 2.0 — First Real World Loan is Financed on Maker ... - Medium.” 21 Apr. 2021, <https://medium.com/centrifuge/defi-2-0-first-real-world-loan-is-financed-on-maker-fbc24675428f>. Accessed 10 May. 2021.

Recommended Readings

1. Evaluating DeFi Lending Protocols
<https://messari.io/article/a-closer-look-at-defi-lending-valuations>
2. How to Assess the Risk of Lending
<https://newsletter.banklesshq.com/p/how-to-assess-the-risk-of-lending>
3. Dashboard on Lending Protocols
<https://terminal.tokenterminal.com/dashboard/Lending>

CHAPTER 6: DECENTRALIZED STABLECOINS AND STABLEASSETS

In our *How to DeFi: Beginner* book, we established that stablecoins are an essential part of the crypto ecosystem. As of 1 April 2021, the total market capitalization for stablecoins is \$64.5 billion - a whopping 12 times increase since the previous year.

As both institutional and retail investors flock to the crypto markets, the demand for stablecoins will continue to flourish. This is unsurprising as stablecoins are non-volatile assets that enable the global transfer of value.

Our beginner's book covered some of the key differences between a centralized stablecoin, Tether (USDT), and Maker's decentralized stablecoin, Dai (DAI). In this chapter, we will be looking at some of the shortcomings of Tether and Dai, before introducing other forms of decentralized stablecoins.

Centralized Stablecoin

Tether (USDT)



USDT (previously known as RealCoin) is a centralized stablecoin that first began trading on the Bitfinex exchange in 2015. Being the first stablecoin in the market, it has a strong first-mover advantage and has consistently maintained its position as the market leader for stablecoins. As of April 2021, Tether has a market capitalization of \$40 billion, representing over 66% of the total stablecoin market share.

Tether maintains its \$1 peg through a 1:1 collateral system. By holding cash as reserve collateral, an equal amount of USDT is then issued. In theory, this is a reliable and straightforward method to ensure USDT maintains its peg. After all, it is a reiteration of the Gold Standard, where the US Dollar used to be backed by Gold in the previous century.

However, the problem lies with Tether's non-transparent issuance process. Because of the centralized nature of its issuance, many people in the crypto community are skeptical that Tether has the reserves it claims to have. In March 2019, skeptics were partially vindicated when Tether modified its policy statement to include loans to affiliate companies as part of its collateral reserves.²⁷

Over the years, multiple government agencies have opened up investigations into Tether's practices. In February 2021, the New York Attorney General's investigation into Bitfinex and Tether's internal finances concluded with no formal charges. However, the New York Attorney General did impose a

²⁷ Coppola, F. (2019, March 14). *Tether's U.S. Dollar Peg Is No Longer Credible*. Forbes. <https://www.forbes.com/sites/francescoppola/2019/03/14/tethers-u-s-dollar-peg-is-no-longer-credible/?sh=43c247aa451b>.

collective fine of \$18.5 million to settle charges where Bitfinex commingled client and corporate funds.²⁸

The Tether saga is a continuing one and unlikely to settle anytime soon. More recent events have lent further credence to its legitimacy, such as Coinbase's listing of Tether. Bitfinex's CTO, Paolo Ardoino has also reiterated that Tether is registered and regulated by FinCEN.²⁹

Decentralized Stablecoin

DAI



Maker is one of DeFi's first attempts at a Decentralized "Central Bank". As a lending protocol, DAI is issued as a byproduct of borrowing demand when collateral (usually ETH) is deposited into Maker vaults.

These vaults are overcollateralized (generally 150% and above except stablecoins), which helps guard against black swan events where the value of the collateral assets drops significantly. Maker helps regulate the supply of DAI by controlling interest loan rates to influence buyers' and borrowers' behavior.

²⁸ Browne, R. (2021, February 23). *Cryptocurrency firms Tether and Bitfinex agree to pay \$18.5 million fine to end New York probe*. CNBC. <https://www.cnbc.com/2021/02/23/tether-bitfinex-reach-settlement-with-new-york-attorney-general.html>.

²⁹ Mike Novogratz Doubts Dogecoin's Future — 'No Institution Is Buying DOGE, & Russian Lawmakers Move to Allow Crypto Payments Under Contracts. (2020, December 31). *Bitfinex CTO: Tether Is Registered and Regulated Under FinCEN- USDT Not Next Target of the US SEC – Altcoins Bitcoin News*. Bitcoin News. <https://news.bitcoin.com/bitfinex-cto-tether-is-registered-and-regulated-under-fincen-usdt-not-next-target-of-the-us-sec/>.

The problem with this setup is that over-collateralization limits capital efficiency, making it difficult for DAI to scale with demand. The arbitrage mechanics used to reduce the price of DAI require higher capital to redeem more DAI.

For example, ETH Vault-A, which has a collateralization ratio of 150%, would require a borrower to fork up another \$1.50 to mint 1 DAI. This has led to scenarios where DAI's price increased above the \$1 peg as it could not keep up with demand, such as the notorious Black Thursday run and Compound's liquidity mining launch.³⁰

Maker has attempted to address these issues through multiple methods, such as its Peg Stability Module solution. However, it is clear that demand for DAI still scales with demand for leverage, rather than demand for a more decentralized medium of transaction.

How do we resolve the stablecoin issue?

Each stablecoin has its own set of issues and is not strictly limited to Tether or DAI. At the heart of the problem is balancing DeFi's ethos of decentralization against creating a currency that can reliably maintain its peg.

Although centralized stablecoins are effective, relying on them requires trust in an incorruptible central entity and offers exposure to the same systemic risks that plague traditional finance. On the other hand, although DAI may be somewhat decentralized, it is still capital-inefficient and cannot meet current market demand.

Several decentralized stablecoin protocols have since emerged, looking to improve decentralization, price stability, and capital efficiency. We call these protocols algorithmic stablecoins.

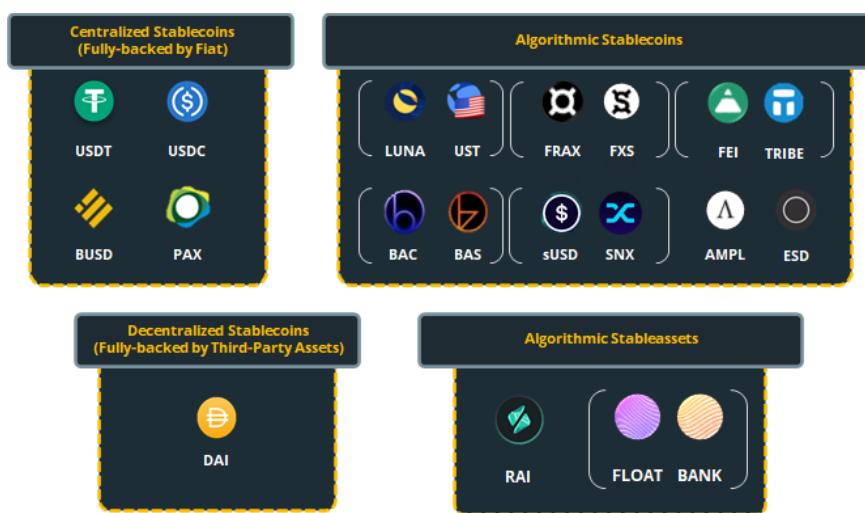
³⁰ Avan-Nomayo, O. (2020, May 4). *MakerDAO Takes New Measures to Prevent Another 'Black Swan' Collapse*. Cointelegraph. <https://cointelegraph.com/news/makerdao-takes-new-measures-to-prevent-another-black-swan-collapse>.

What are Algorithmic Stablecoins and Stableassets?

As its name suggests, algorithmic stablecoins utilize algorithms to control the stablecoin's market structure and the underlying economics. You may think of algorithmic stablecoins as an automated Federal Reserve, where instead of human-made decisions, pre-programmed code executes specific actions to control and influence the price.

While most algorithmic stablecoins are pegged to the US Dollar, some are not, and some even have a floating peg, thereby creating a new type of asset that we would classify as algorithmic stableassets. Unlike algorithmic stablecoins, algorithmic stableassets could be seen as another form of collateral rather than units of accounts.

If we were to categorize the stablecoins or stableassets market in its current state, the following illustration provides a concise overview:



Source: CoinGecko Q1 2021 Report

*List is non-exhaustive

There are two ways of categorizing the various decentralized algorithmic stablecoins:

- a) Has no collateral (ESD, AMPL, and BAC)
- b) Is partially/fully collateralized by their own native token (FRAX, sUSD, and UST)

For the sake of categorizing stablecoins, we have bucketed UST and sUSD as algorithmic stablecoins because it is backed by its native token, unlike DAI, which is backed by third-party assets such as ETH and USDC. This is a key criterion because reliance on natively issued assets as collateral creates recursive value, requiring algorithmic functions to regulate the price.

Algorithmic stablecoins can be further broken down into different categories - the main sub-categories are rebase and seigniorage models.

Rebase Model

A rebase model controls the price by changing the entire supply of the stablecoin. Depending on whether the price of the stablecoin is above or below the intended peg, the protocol will automatically increase or decrease the supply in every holder's wallet over a fixed period.

The reasoning for this is that by forcefully controlling the supply, the price of the stablecoin can be influenced based on a simple inflationary/deflationary economic theory.

The pioneering protocol for this model is Ampleforth (AMPL), dating as far back as 2019.

Ampleforth

Ampleforth

Every 24 hours, the entire circulating supply of Ampleforth (AMPL) is either proportionally increased or decreased to ensure the price remains at \$1.³¹ If the price of AMPL is trading 5% or more above the target of \$1, the rebase will expand supply to wallets holding AMPL. If the price of AMPL is trading 5% or more below the target of \$1, the rebase will decrease units in wallets holding AMPL.

Every wallet holder will be affected, but they will retain the same market share as before. The rebases, both positive and negative, are non-dilutive because they affect all AMPL balances pro-rata.

Since rebasing occurs at fixed intervals, users can time their trades to purchase or sell AMPL right before rebasing to increase the value of their holdings.

Seigniorage Model

A seigniorage model controls the price by introducing a reward system that influences market dynamics. If the price is above the peg, new tokens are minted and given to participants who provide liquidity or have tokens staked.

If the price is below the peg, tokens stop being minted, and mechanics are introduced to reduce the supply. Users may purchase coupons that burn the underlying tokens to remove them from the supply. These coupons may be redeemed for more tokens in the future, but only when the price returns or exceeds the intended peg.

³¹ Ampleforth. (n.d). <https://www.ampleforth.org/basics/>.

There are three basic iterations of this model:

Empty Set Dollar



Empty Set Dollar (ESD) is a single-token seigniorage model. As the term implies, there is only a single token in this model. Users provide liquidity or stake in the Decentralized Autonomous Organization (DAO) through the protocol's native token - the ESD token effectively acts as both a stablecoin and governance token.³²

At the start of every epoch, the system will measure the Time-Weighted Average Price (TWAP). If the TWAP is above \$1, the protocol will enter an inflationary phase and mint tokens as rewards for DAO stakers and liquidity providers. Conversely, if the price falls below \$1, the protocol enters a contractionary phase where users will stop receiving any rewards.

During the contractionary phase, users may purchase coupons by burning ESD to earn a premium of up to 45% if the protocol enters an expansion phase again. However, coupons only last for 30 days, meaning coupon buyers risk getting nothing if the system stays in a contractionary phase for over 30 days. Epochs in ESD lasts for 8 hours.

³² *Empty Set Dollar Basics*. Empty Set Dollar Basics – Empty Set Dollar. (n.d.). <https://docs.emptyset.finance/faqs/basics>.

Basis Cash



Basis Cash is a dual-token seigniorage model. As the term implies, there is an additional token known as the share token. Basis Cash's stablecoin is Basis Cash (BAC), while its share token is Basis Share (BAS).³³

Like ESD, Basis Cash relies on a TWAP mechanism that will mint or stop minting BAC, depending on whether BAC's price is above or below \$1. When BAC is above \$1, the protocol enters an expansionary phase where users may receive newly minted BAC from the Boardroom DAO by staking BAS.

When BAC is below \$1, the protocol enters a contractionary phase where no new BAC will be minted to BAS stakers in the Boardroom. Basis Bonds (similar to ESD coupons) become available for purchase and are priced at $(BAC)^2$. Purchasers of Basis Bonds will get to redeem BAC when the protocol enters an expansion phase again.

Basis Cash epochs last for 24 hours, and unlike ESD coupons, Basis Bonds do not have an expiry date.

³³ Dale, B. (2020, November 30). '*Basis Cash*' Launch Brings Defunct Stablecoin Into the DeFi Era. CoinDesk. <https://www.coindesk.com/basis-cash-algorithmic-stablecoin-launch-defi-rick-morty>.

Frax Finance



Frax borrows principles from the seigniorage model to create its own unique model whose stablecoin (FRAX) is backed by two types of collateral - fiat-backed centralized stablecoin (USDC) and its native share token, Frax Share (FXS).³⁴ Although Frax currently employs fiat stablecoins as collateral, the protocol does intend to eventually fully diversify into decentralized collateral.

Unlike Basis or ESD, Frax takes a fractional collateralization approach. Frax has an adjustable collateral ratio controlled by a Proportional Integral Derivative (PID) controller. The collateral ratio is adjusted according to a growth ratio that measures how much FXS liquidity there is against the overall supply of FRAX.

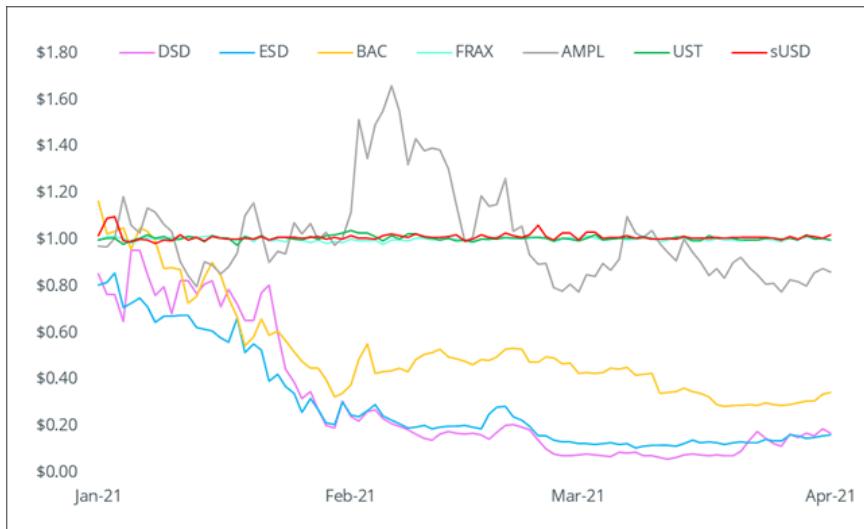
Frax stablecoins (FRAX) can be minted and redeemed from the protocol for \$1 worth of value. This incentivizes arbitrageurs to purchase or mint FRAX, bringing the price back to its peg. On the other hand, FXS accrues fees, seigniorage revenue, and excess collateral value. Notably, Frax launched veFXS which gives a portion of the Algorithmic Market Operations Controller (AMO) profits to FXS stakers, similar to how veCRV receives a portion of transaction fees on Curve Finance.

When FRAX is above \$1, the PID lowers the collateral ratio by one step, and when FRAX is below \$1, the PID increases the collateral ratio by one step. Both the refresh rate and step parameters can be adjusted through the FXS governance.

As of 8 May 2021, FRAX is 85.25% backed by USDC and 14.75% backed by FXS.

³⁴ Introduction. Frax Finance. (n.d.). <https://docs.frax.finance/overview>.

How has Algorithmic Stablecoins fared so far?



Source: CoinGecko Q1 2021 Report

Throughout the last quarter of 2020, we have seen many algorithmic stablecoins launched in the market. Many of these stablecoins use the seigniorage model.

Despite the crypto community's excitement, most of these protocols have failed. If we look at the historical price of the top-7 algorithmic stablecoins by market capitalization, only UST, sUSD, and FRAX have managed to maintain their peg to the US Dollar.

Although each protocol has its own design mechanics and unique reasons for failure, we are able to make some general observations on the issues with seigniorage-type stablecoins.

Historically, the fiat currency system was easier to implement because nations have centralized powers. For example, monarchies and governments have the monopoly to mint coins and guarantee their value. Contrast this with the stablecoin market where new protocols have to do the same overnight based on an experimental product, in a decentralized environment heavily saturated with competition and established incumbents.

To overcome this, algorithmic stablecoins offered eye-watering incentives in the form of extremely high liquidity mining rewards. The problem with this approach is that it primarily attracted speculators looking to make a quick flip, birthing a new species of yield farmers known as Algo-Farmers.

Algo-Farmers had one objective - look for new algorithmic stablecoin protocols before anyone else, farm the native stablecoin by providing liquidity, and exit the system once other Algo-Farmers start pouring in. Therefore, distribution was centralized within the first group of Algo-Farmers, and dumped onto latecomers. Algorithmic stablecoin launches effectively became a game of musical chairs as profiteers had little incentive to commit and proactively contribute to a single community, quickly moving on to other algorithmic stablecoins clones.

Unsurprisingly, the supply shocks led to massive price swings. Most algorithmic stablecoins were designed with a somewhat cooperative community in mind, but that failed to incentivize users to support the system during contractions or even counteract price manipulations by bigger players.

On top of that, algorithmic stablecoins are mostly traded on Automated Market Makers like Uniswap, which amplified volatility due to its constant product market characteristics. During the deflationary stage, liquidity providers are also disincentivized from supplying liquidity because of impermanent loss risk, leading to further volatility.

Why has FRAX succeeded?

While sUSD, UST, and FRAX have their own reasons for their successes, we will focus on FRAX because it has the best peg retention to date, with an average price of \$1.001 throughout Q1 2021. Here are some of the reasons why we think they have been successful so far:

1. Frax is partially collateralized by USDC, which instills community confidence in the system to maintain its peg. As of 8 May 2021, 85.25% of FRAX is backed by USDC.

2. Frax keeps the collateralization ratio flexible, addressing market demands for pricing FRAX at \$1.
3. Collateral is redeployed elsewhere to earn interest. This helps to bring in external revenue and keeps the protocol afloat, which is then used to buyback and burn FXS.
4. Price volatility of FRAX is shifted to FXS because of its buyback and burn mechanic.

While these are just simplified reasons, they will still need to be continuously reconsidered, especially in times of crisis when the price of FXS significantly drops and when liquidity mining rewards eventually stop.

The Next Generation of Algorithmic Stablecoins and Stableassets

While FRAX has largely been successful, one could argue that it is not truly decentralized. This is because it is still partially collateralized by USDC, which is backed by cash reserves held by a centralized entity (CENTRE Consortium - with Coinbase and Circle as co-founding members).

New entrants are trying to avoid this dilemma. They are still gravitating towards the collateralized system but have their distinctive takes, making it difficult to categorize under the existing umbrella of algorithmic stablecoins. We will cover three examples.

Fei Protocol



Fei launched at the end of Q1 2021. Much like Frax, Fei uses a partially collateralized system but is instead backed purely by ETH. Fei's stablecoin (FEI) is pegged to \$1, which is underpinned by innovative concepts to maintain their peg and ensure the overall financial stability of the protocol.³⁵

Rather than borrow collateral, Fei introduces a mechanism known as the Protocol Controlled Value (PCV). Essentially, Fei purchases ETH from users through newly minted FEI. Fei will then use the ETH to support their collateral-backed liquidity pools. Other common use cases include governance treasuries and insurance funds. During the initial launch, Fei allocated 100% of the PCV funded by the ETH bonding curve to a Uniswap pool using an ETH trading pair.

When the price of FEI is above \$1, the protocol allows users to mint new FEI directly from the system at a discounted price (similar to FRAX) using ETH as payment. Traders may then arbitrage the price down until the price reaches its \$1 peg. When the price of FEI is below \$1, the protocol taxes FEI sellers (whose tax is then burned and removed from the supply), and awards extra FEI to buyers (on top of their initial purchase). The trading algorithm ensures that the tax amount exceeds the amount that buyers would receive.

In emergencies where the price of FEI is below the peg for an extended period, FEI may withdraw their PCV-backed liquidity from Uniswap and

³⁵ Fei Protocol. (2021, January 11). *Introducing Fei Protocol*. Medium. <https://medium.com/fei-protocol/introducing-fei-protocol-2db79bd7a82b>.

buy FEI from the market. At the same time, FEI is also burned. Once the peg is restored, Fei will resupply the remaining liquidity back into Uniswap.

Fei also has a native governance token (TRIBE) which will eventually become the foundation for a DAO. In the future, TRIBE holders will be able to decide on adjusting the PCV allocation and adding/adjusting bonding curves.

Reflexer



Unlike other algorithmic stablecoins, Reflexer's native token (RAI) is not meant to be a fixed-peg stablecoin. Launched in Q1 2021, RAI's intended purpose is to become stable collateral and replace existing collateral assets such as ETH or BTC, which are naturally volatile. RAI uses an ETH-based overcollateralized model and has a floating peg that was initially set to \$3.14.

Reflexer uses managed-float regime principles, similar to how central banks operate.³⁶ Since prices constantly fluctuate, Reflexer designed a system where market interactions between RAI minters and traders (RAI holders) are incentivized to chase RAI's redemption price (floating peg) in order to keep the price of RAI relatively stable.

To mint RAI, users need to deposit ETH as collateral with a minimum of 145% collateralization ratio. Users are then charged a stability fee (borrowing interest rate). At the time of writing (May 2021), the stability fee is 2% per annum. However, it is variable and may be amended through a governance vote.

³⁶ Ionescu, S. (2020, October 29). *Introducing Proto RAI*. Medium.
<https://medium.com/reflexer-labs/introducing-proto-rai-c4cf1f013ef>.

When the price of RAI is above the floating peg, the system lowers the peg. This allows users to mint more RAI and sell it back for ETH for a higher return. When the price of RAI is below the floating peg, the system raises the peg. This makes borrowing more expensive and incentivizes RAI minters to repay their RAI loans, thereby removing RAI from circulation and driving the price up.

In emergency situations (Settlement), the protocol shuts down and only allows both RAI minters and RAI holders to redeem ETH collateral from the system at the current redemption price.

Reflexer also has another native token (FLX) which acts as the lender of last resort, governs certain functions, and allows users to stake it in a pool that protects the system. There are also debt auctions, which are created to repay FLX in exchange for RAI - the RAI that is received by an auction will be used to eliminate bad debt from the system. On a longer time horizon, FLX is intended to be an “ungovernance” token, progressively automating the system over time and minimizing governance.

Float Protocol



Float is quite similar to FRAX, where it uses a two token seigniorage model and is partially collateralized by ETH. However, unlike FRAX, Float’s native asset (FLOAT) has a floating rate but has an initial peg price of \$1.61. Float’s share token (BANK) also acts as both a governance token and regulating mechanism to support FLOAT’s price.³⁷

³⁷ Float Protocol. (2021, March 22). *Announcing Float Protocol and its democratic launch*. Medium. <https://medium.com/float-protocol/announcing-float-protocol-and-its-democratic-launch-d1c27bc21230>.

Float uses a similar mechanism as Fei's PCV where users may only acquire newly-minted FLOAT from the protocol. However, Float sells FLOAT through a Dutch auction where prices are listed at the highest possible price and descend downwards towards the minimum (reserve) price. To acquire FLOAT, users must pay with a combination of BANK and ETH. The asset payment ratio depends on the overall demand for FLOAT and the value of ETH in the basket.

If the basket is in excess, ETH is used to purchase FLOAT at the target price, and BANK is used to mint FLOAT. ETH (or any other future trading assets) earned from the Dutch auction is then stored as collateral in the protocol's collateral vault (Basket). BANK is burned whenever a user mints FLOAT.

When the price of FLOAT is above the floating peg, any user can start an auction. Initially it can only be started if a minimum of 24 hours has passed but this will be removed in the future once the team is satisfied that users are accustomed to the auction feature. Once an auction starts, the system mints and sells new FLOAT, starting at market price plus an added premium. Price will lower over time until it reaches the target price.

When the price of FLOAT is below the floating peg, the protocol offers to buy FLOAT from the market in the form of a reverse Dutch auction. This is where Float offers buyers what bids it would accept at incremental prices. FLOAT is bought with both ETH and freshly minted BANK.

Float's initial collateral ratio (Basket Factor) was set at 100% at launch but may be amended through governance voting. During emergencies, assets stored in the basket can be used to support the price of FLOAT if it is below its peg.

How will these new Algorithmic Stablecoins and Stableassets fare?

Older algorithmic stablecoins prioritized capital efficiency over everything else. Pegs were fixed and purely reliant on individualistic game theory mechanics, while untested bootstrapping methods were also prevalent.

Stablecoins were known for their peg to the US Dollar, but newer developments in the space have changed that very definition. We now have a new class of assets known as algorithmic stableassets.

We consider FLOAT and RAI as algorithmic stableassets primarily because they have a floating peg. Regardless, the focus should be on whether these assets can maintain a stable price or not. To answer this question, we considered three main factors.

I. Collateralization

Newer algorithmic stables are adopting a more conservative approach where collateralization takes precedence over capital efficiency.

Fei's PCV approach utilizes "liquidity collateralization" where collateral is automatically diverted to Fei's Uniswap liquidity pools. Governance voting (through the TRIBE native token) will allow users to control the PCV ratio, which is effectively the collateral ratio.

This is similar to Float's Basket Factor. However, Float has the added advantage of a floating peg and a share token (BANK) which is necessary for acquiring FLOAT. This increases its value and helps 'store' extra volatility into BANK as opposed to FLOAT. BANK can be used as collateral, though it would also be possible for Fei to collateralize TRIBE, albeit with less inert value since it possesses less utility. In other words, both Float and Fei let market forces decide on the ideal collateral ratio (similar to FRAX).

Contrast this with Reflexer's RAI, which is overcollateralized and is less prone to black-swan events. There is a minimum collateralization requirement but no maximum. Indeed, one could argue that Reflexer has a strong likelihood for success since it is a fork of Maker's Multi-Collateral Dai and managed to retain its peg after Black Thursday (despite the length of time it took). Furthermore, RAI does not confine itself to a fixed peg, thus giving it greater flexibility.

II. Trader Incentives/Disincentives

Algorithmic stablecoins and stableassets are designed to influence market behavior to help maintain its peg price. Older generations focused on rewarding “correct” user behavior (i.e., arbitrageurs). This still carries on for newer algorithmic stablecoins and stableassets where all three possess similar minting reward mechanics when the price is above the peg. There is, however, a noticeable difference in approach during the deflationary phase.

Newer protocols are incorporating “negative reinforcement” tactics. Fei penalizes sellers with a trading tax whenever FEI is sold below the peg. Reflexer indirectly raises the borrowing rate through the peg raise, which encourages borrowers to repay their loans (similar to Maker). Float is slightly different as it lets users’ battle’ it out in their reverse Dutch Auction. Rather than penalizing or rewarding users, Float lets market forces decide.

III. Emergency Powers

Perhaps the most interesting development is the push for stronger protocol powers. Each protocol’s system design has built-in functions to protect its market when its native asset significantly devalues. One could draw parallels to traditional finance where regulators or centralized financial authorities step in during financial crises.

Fei essentially cuts off access to liquidity by removing their PCV-backed liquidity from Uniswap pools - similar to how a country

might impose limits on bank withdrawals. Fei will then sell off its assets and offer to buy back excess FEI from the market. Float executes a similar tactic, except purchases are financed from the Basket. Reflexer halts all borrowings and only allows repayment of loans.

Associated Risks

We cannot emphasize enough that algorithmic stablecoins are still very much in the experimental phase. Protocols are still trying to figure out how to launch successfully without massive price swings.

Many algorithmic stablecoins protocols are also heavily reliant on competent arbitrageurs to maintain the price peg. If you are unsure how the protocol works, you would be at a severe disadvantage if you try to compete with savvy arbitrageurs (or even bots).

Algorithmic stablecoins require a strong community that believes in the project's fundamentals. More often than not, short-term profiteers will leverage their capital reserves to control and manipulate the price. In a decentralized market, only a cooperative community with strong underlying mechanisms can overcome this dilemma.

In other words, you would need to commit significant time and resources to understand each project. Only then can you decide whether it can compete with the many alternative stablecoins/stableassets that already have an established market presence.

Notable Mentions

-  **Empty Set Dollar v2 (ESD)**
ESD is migrating towards a dual-token stablecoin model by introducing a new token, ESDS. ESD v2 (also known as Continuous ESD) will be quite similar to Frax in having a partially collateralized stablecoin by incorporating a bank reserve backed by USDC.

Together with ESDS, these two new features will hopefully help mitigate the volatility of ESD tokens.

-  **Dynamic Set Dollar v2 (DSD)**
While most stablecoin protocols are focussed on a partial or fully collateralized model, DSD (a fork of ESD) believes that this detracts from the ethos of decentralization. Despite its initial failure, DSD has updated its model by introducing a new token, CDSD, which is partially redeemable 1:1 for DSD tokens. The idea is to transfer the volatility of DSD tokens onto CDSD - similar to Frax's model, but without any collateral.
-  **Gyroscope (GYR)**
Gyroscope's mechanics is an amalgamation of multiple algorithmic stablecoin protocols with its own twist. GYR is over-collateralized and backed by multiple assets split into individual vaults (similar to Maker). Like most algorithmic stablecoins, there are arbitrage mechanics but with the addition of a complementary leveraged loan mechanism. During times of crisis, users will get a more favorable redemption rate the longer they wait to pay back their loans.
-  **TerraUSD (UST)**
Similar to DSD v2, UST is an uncollateralized dual token model (along with LUNA) and fully relies on arbitrage to maintain its \$1 peg. At the time of writing, UST is one of the only successful algorithmic stablecoins to maintain a stable price peg - likely due to strong demand and a flourishing ecosystem (Terra) that incorporates the mining network into its price stability mechanisms.

Conclusion

Algorithmic stablecoins are effectively DeFi's take on replacing a central bank, while algorithmic stableassets are DeFi's way to emulate the Gold Standard and create reliable digital collateral. In traditional finance, a

successful monetary system requires a competent and independent financial authority. In DeFi, competency is sourced from pseudo-anonymous individuals who are incentivized to collaborate and act rationally.

Successful algorithmic stablecoins and stableassets require longer time-frames to prove themselves, especially during times of crisis. Neither short-term incentives nor short-term speculation is sustainable in and of itself - they need to become more than just a thought experiment and offer economic utility through widespread adoption. It will be interesting to monitor how the algorithmic stablecoins and stableassets in this chapter will perform over the coming years.

Recommended Readings

1. Understanding Risk of Rebase Tokens Through Smart Contract Analysis
<https://www.coingecko.com/buzz/understanding-risk-of-rebase-tokens-through-smart-contract-analysis>
2. Understanding Fei, Float and Rai
<https://medium.com/float-protocol/float-and-the-money-gods-5509d41c9b3a>
3. Exploring the Key Success Factors for Algorithmic Stablecoins
<https://messari.io/article/the-art-of-central-banking-on-blockchains-algorithmic-stablecoins>
4. Deeper dive into Rebase and Seigniorage Models
<https://insights.deribit.com/market-research/stability-elasticity-and-reflexivity-a-deep-dive-into-algorithmic-stablecoins/>
5. Impact of Uniswap on Algorithmic Stablecoins
<https://medium.com/stably-blog/what-uniswaps-liquidity-plunge-reveals-about-stablecoins-4fcbee8d210c>

CHAPTER 7: DECENTRALIZED DERIVATIVES

The acceptance of digital assets have progressed into the creation of sophisticated financial products for users and traders. Currently, the usage of crypto derivatives are more commonplace at the centralized platforms like Binance Futures, Deribit, FTX, and Bybit.

With the growth of decentralized derivatives platforms, traders can now trade crypto derivatives in a trustless manner too. In this chapter, we will be going through decentralized derivatives in three distinct sections - decentralized perpetuals, decentralized options, and synthetic assets.

Decentralized Perpetuals

As one of the most popular derivatives in the crypto-space, perpetual swaps enable users to open a leveraged position on a futures contract with no expiration date. Previously, perpetuals were only available on centralized exchanges, however decentralized platforms such as Perpetual Protocol and dYdX have since paved the way for the wider DeFi community to gain access to leveraged positions while being fully in control of their own funds.

Perpetual Protocol



Perpetual Protocol is a decentralized protocol that offers perpetual contract trading, allowing users to open up to ten times leverage long or short positions on various cryptoassets. To achieve this, Perpetual Protocol uses a unique approach of virtual Automated Market Makers (vAMMs).

Functioning similarly to Uniswap and Balancer's AMMs, traders can execute transactions directly through the vAMMs. The main difference lies with the “virtual” part.

In conventional AMMs, assets are stored within the smart contract, and the exchange price for each asset is determined through a specific mathematical function. vAMMs in Perpetual Protocol do not store any assets whatsoever.

Instead, the real assets are stored in a smart contract vault, denominated in USDC, which becomes the collateral for users to open a leveraged position. The total amount of funds in the vault essentially forms the cap for traders' profits. Each perpetual contract will have its own specific vAMM, but all of them are protected under the protocol's insurance fund.

Perpetual Protocol is able to offer increased trading speed and minimal gas trades with the use of the xDai chain for trade execution. With vAMM, users will have access to high liquidity and low slippage for their trading needs.

As with all forms of perpetual contract trading, funding rates and liquidation ratios are crucial aspects of Perpetual Protocol. Funding rates are settled on an hourly basis, while liquidation ratios are set at 6.25% of posted margin. This means that traders with positions that fall below the 6.25% margin ratio will face the risk of being liquidated by keeper bots. Keeper bots will earn 20% of the liquidated margin, while the remainder will be sent to the protocol's insurance fund.

The protocol has its own native PERP token, which primarily serves as a governance token for the platform. PERP token holders receive voting power in proportion to their holdings. Additionally, they may stake their PERP for a fixed period to receive even more PERP and a share of the protocol's transaction fees in USDC.

This fixed period, known as an epoch, lasts for seven days. Token holders may not withdraw their funds until the end of each epoch. Transaction fees are claimable immediately after, while PERP rewards are locked for up to 6 months. Stakers get to enjoy zero impermanent loss, but they would still be exposed to the price volatility of the PERP token itself.

That's all you need to know about Perpetual Protocol. As of April 2021, trading is live on the Ethereum and xDai mainnet, so you can go ahead and give it a spin.

dYdX



dYdX is a decentralized exchange protocol for lending, borrowing, spot trading, margin trading, and perpetual swap trading. One of the first projects to specialize in decentralized perpetual, dYdX supports three assets for spot and margin trading - ETH, USDC, and DAI. For perpetual swaps, 11

different contracts are available for trading, including BTC, ETH, AAVE and LINK.

dYdX shares some common characteristics with other lending and borrowing platforms such as Aave and Compound - allowing users to deposit their assets to earn interest or use their deposited assets as collateral for borrowing. However, dYdX sets itself apart by incorporating margin trading on ETH, with up to five times leverage, using either DAI or USDC. Users can also utilize up to ten times leverage on perpetual contracts trading on dYdX.

Lending on dYdX is flexible and automatically matched to borrowers, so there is no waiting period before you can start earning interest on deposits. Interest payments are compounded every time a transaction is made using that asset.

Interest rates are dynamically updated based on the level of utilization - higher utilization leads to higher interest rates for lenders. For borrowers, an initial collateralization ratio of 125% is required, while a minimum ratio of 115% needs to be maintained to prevent automatic liquidation.

dYdX offers spot traders similar functions as centralized exchanges such as market, limit, and stop orders. Trading fees for margin or spot positions are limited only to takers, where the amount charged is either 0.3% or the variable gas costs at the time, whichever is higher. To minimize gas cost, traders should pay attention to order size - the platform charges additional fees on smaller orders to pay for gas fees to complete the transaction.

For dYdX's perpetual markets, all contracts are collateralized in USDC. However, each contract uses different oracles, order sizes, and margin requirements. Funding rates are continuously charged each second for as long as a position remains open. The rates are recalculated every hour and are represented as an 8-hour rate, similar to Binance Futures. Unfortunately, perpetual contracts on dYdX are not available for United States residents.

In Q1 2021, dYdX partnered with Starkware to build a Layer-2 trading protocol, allowing for faster and cheaper transactions. Using a scalability

engine known as StarkEx, trades will be matched off-chain using zero-knowledge rollups (zK-Rollup) and settled on the Ethereum mainnet. Users can now access the Layer-2 markets by generating a Stark Key, used to identify your Layer-2 account on dYdX and to send a transaction to register the account on-chain.

Comparison between Perpetual Protocol and dYdX (Layer 1)

Factor	Perpetual Protocol	dYdX (Layer 1)
Contracts supported	BTC, ETH, YFI, LINK, DOT, SNX	BTC, ETH, LINK, AAVE, UNI, SOL, SUSHI, YFI, 1INCH, AVAX, DOGE
Exchange Model	Virtual AMM	Orderbook
Maximum Leverage	10x	10x
Initial Margin	10%	10%
Maintenance Margin	6.25%	7.5%
Funding Rate	Every hour	Every second
Transaction Fees	0.1% on notional	Maker: -0.025% Taker: Higher of 0.2% or gas costs

In terms of contract specifications, both platforms offer highly similar leverage options and margin requirements for their respective markets. Perpetual Protocol offers a larger variety of assets and a much forgiving funding rate for open positions, bringing in an average of \$60 million in daily trading volume compared to \$15 million done by dYdX on Layer 1 in the first quarter of 2021.³⁸ On the other hand, dYdX takes the “exponential” approach by continuously benefiting its position holders, even if the higher fees may discourage smaller traders.

³⁸ (n.d.). Perpetual Swaps Trade Volume - The Block. Retrieved April 29, 2021, from <https://www.theblockcrypto.com/data/decentralized-finance/derivatives/perpetual-protocol-trade-volume>

In our view, dYdX may have the potential to compete with Perpetual Protocol after the introduction of both Layer-2 technology and more markets for popular tokens such as Aave and Uni, and on a lower fee basis.³⁹

Notable Mentions

-  **Futureswap**

Futureswap is a decentralized perpetual exchange that allows users to trade up to 10x leverage on any ERC-20 pairs.

-  **MCDEX**

MCDEX is an AMM-based decentralized perpetual swap protocol that currently uses the second iteration of their Mai Protocol. Any user can create a perpetual market, as long as there is a price feed for the underlying asset and ERC-20 tokens to use as collateral.

-  **Injective Protocol**

Powered by the Injective Chain, this Layer-2 derivatives platform supports a fully decentralized order book and bi-directional token bridge to Ethereum. As of April 2021, it is still in testnet.

Decentralized Options

Options have long been a staple of traditional finance, offering buyers the opportunity to bet on price movements in either direction to hedge the value of their assets or amplify their returns with minimal capital. As DeFi continues to make waves throughout the crypto industry, it is only natural that decentralized options protocols come about as well.

Crypto users have historically traded options on centralized exchanges such as Deribit, but there is an inherent demand for decentralized options protocols. In this chapter, we will be looking at two leading decentralized options protocols, Hegic and Opyn.

³⁹ (2021, April 12). UNI, AAVE Markets now live - dYdX. Retrieved April 20, 2021, from <https://dydx.exchange/blog/markets-01>

Hegic



Hegic is a decentralized on-chain protocol that allows users to purchase American call and put options on ETH and WBTC. Users may also sell options by being liquidity providers to earn premiums. Using the platform's interface, users can customize the terms of the options they want to purchase, such as the strike price and expiry date.

Options' prices will be automatically calculated once the terms are selected, including a 1% settlement fee on the option sizes purchased. Although the options are non-tradeable, users can exercise them at any time since liquidity is locked on the option contract.

Hegic operates using a liquidity pool model. In other words, users pool their funds together and use it as collateral for underwriting all options sold. As of April 2021, there are two separate pools for ETH and WBTC. Liquidity providers lock up either ETH or WBTC and receive a certain amount of Write tokens according to the asset provided.

Write tokens represent the provider's claims on the premiums paid by users to purchase options. Although anybody can purchase options from the liquidity pools, the maximum purchase limit for each pool is set at 80% of the total underlying collateral.

An interesting thing to note about Hegic is that you can get rewarded for using the platform. HEGIC has a liquidity mining program that rewards users who stake their Write tokens and rewards option buyers based on option size and duration purchased. These rewards come in the form of

rHEGIC tokens, a token claimable for the actual HEGIC token once certain milestones on the Hegic platform are achieved.

HEGIC tokens can also be staked to earn the protocol's fees, where 100% of the settlement fees from both pools are distributed to Hegic Staking Lot owners. To own a lot, you would need 888,000 HEGIC. If that's a little too much, you can delegate any amount you wish at www.hegicstaking.co.

Hegic has amassed more than \$50 million in Total Value Locked and has settled more than \$22 million of trading volume in a single day.^{40,41} With these figures, Hegic remains as one of the top decentralized options protocols.

And that's pretty much it for Hegic! Next, we will be looking at another decentralized options platform called Opyn.

Opyn



Opyn is one of the first decentralized options platforms to launch. The first version, Opyn V1, lets you create tokenized American options in the form of oTokens by locking 100% of the underlying asset as collateral. This ensures that the holders can always exercise the options since it is fully collateralized.

Provided there are enough liquidity providers to lock in their collateral, Opyn can offer a wide range of options on assets such as ETH, WBTC, UNI, and

⁴⁰ (2021, April 10). Hegic Quarterly Report #2. Hegic has achieved steady ... - Medium. Retrieved May 7, 2021, from <https://medium.com/hegic/hegic-quarterly-report-2-6c4170ac82e0>

⁴¹ (n.d.). Options TVL Rankings - Defi Llama. Retrieved May 7, 2021, from <https://defillama.com/protocols/options>

SNX, albeit with a fixed duration and strike price. oTokens can be exercised by sending the strike amount of stablecoins and burning the oTokens in exchange for the underlying asset, or it can be resold to other parties via Uniswap. On top of that, Opyn does not charge any additional transaction or settlement fees.

The second version, Opyn V2 was launched with additional features such as auto-exercise and flash minting, an innovative spin on the existing concept of flash loans popularized by Aave. The latest iteration offers European options through an order book system, similar to Deribit, but is currently limited to Wrapped Ether (WETH) and with a smaller range of strike prices and expiry dates.

Comparison between Hegic and Opyn

How do these two protocols stack up against each other? In the table below, we compare both platforms based on several factors.

Factor	Hegic	Opyn V1	Opyn V2
Option Type	American	American	European
Liquidity Model	Single-asset Pool	Uniswap Pool	Order book
Settlement Type	Cash	Cash	Cash
Collateral Assets	ETH, WBTC	ETH, WBTC, UNI, SNX	WETH, WBTC
Premium Payment	ETH	DAI, ETH, USDC	USDC
Collateral Requirement	100%	100%	100%

Here, we can see that both options platforms need to be fully collateralized by the options writers, but are very different in terms of liquidity models and the number of assets supported.

American options seem to be favored by both protocols due to the degree of flexibility it offers in the fast-paced DeFi space. This is in contrast to European options, which is favored by centralized derivative exchanges such as Deribit. In Opyn V2, thin order books with a smaller selection of strike prices indicate lower demand for products with a heavy time constraint, given the volatile nature of the digital asset space.

Notable Mentions

-  **FinNexus**
FinNexus allows users to create options on almost any asset, as long as there is a reliable price feed. The protocol uses a Multi-Asset Single Pool (MASP) system, which allows for positions on different underlying assets while using only a single asset type as collateral.
-  **Auctus**
Auctus is a DeFi protocol that allows users the ability to perform flash exercises, where they do not need to own the strike tokens to exercise their options. They also offer principal-protected yield farming through their Auctus Vaults and a dedicated section for OTC options trading.
-  **Premia**
Premia offers a secondary marketplace for users to purchase and sell their options. Users can mint, transfer and exercise multiple options types with fewer transactions, saving both time and gas costs.
-  **Antimatter**
Antimatter aims to market itself as the Uniswap of options by providing an exchange for their perpetual options products. Users can receive long or short exposure by purchasing these Polarized option tokens, which can be redeemed without worrying about expiry dates.



- **Siren Protocol**

Through Siren Protocol, users can choose to be a writer or purchaser of options by purchasing either bTokens or wTokens of a particular options series from the SirenSwap Automated Market Maker. bTokens represent the buyer's side, allowing holders to exercise the options, while wTokens represent the writer's side, which will be used to withdraw the collateral or receive payment upon exercise.

Synthetic Assets

Synthetic assets are assets or a mixture of assets that have the same value or effect as another asset. Synthetic assets track the value of underlying assets and allow exposure to the assets without the need to hold the actual asset itself.

Examples of synthetic assets include virtually any trackable assets, from real-world stocks to Ethereum gas prices and even metrics on the CoinGecko website. Users trading these synthetic assets can have financial exposure in these assets without holding any of the actual assets themselves.

In this chapter, we will be comparing two of the largest protocols in the DeFi synthetic assets sector, namely Synthetix and UMA.

Synthetix



The Synthetix logo consists of the word "SYNTHETIX" in a bold, white, sans-serif font, centered within a dark blue rectangular background.

We have covered Synthetix extensively in our *How to DeFi: Beginner's* book, but here's a little recap.

Synthetix is a decentralized platform for minting and trading synthetic assets known as Synths, backed by collateral provided by the platform's users.

Synths allow users to track the value of an underlying asset without holding the actual asset itself. There are two types of Synths - Regular Synths (e.g. sDEFI) and Inverse Synths (e.g. iDEFI). Not all Synths have an inverse counterpart.

Synths can be created for various asset classes such as cryptocurrencies, fiat currencies, commodities, equity indexes, and equities. The prices of the assets are tracked using Chainlink, a decentralized oracle that gathers price feeds from multiple sources.

Synths are created using over-collateralization, a concept similar to Maker in minting Dai. To mint Synths, users would have to stake the Synthetix Network Token (SNX) as collateral. Since the value of SNX can move quickly in either direction, a large collateralization ratio of 500% is required to mitigate liquidation risks.

To maintain a minimum of 500% collateralization ratio, users can burn Synths if they are below the target ratio or mint more Synths if they are above the threshold. Do note that as of January 2021, the only Synth that users can mint is sUSD.

Synths are mainly traded on the Synthetix Exchange, a decentralized exchange that does not rely on order books but on user liquidity. Synthetix Exchange allows users to trade directly against a smart contract that maintains constantly adequate liquidity, which reduces the risk of slippage. This is particularly useful for large transactions that would otherwise incur significant price slippages on other exchanges.

To incentivize staking and minting of Synths, users have the chance to receive exchange fees and SNX staking rewards. Fees generated from trading on the Synthetix Exchange are sent to a pool, where SNX stakers can claim their proportion of the fees collected. Stakers can also claim rewards of SNX tokens as long as their collateralization ratio does not fall below the current threshold.

Now that you have had a refresher on Synthetix, let's dive into what UMA is all about.

UMA



UMA or Universal Market Access is a decentralized protocol for creating and enforcing synthetic assets on the Ethereum network. UMA provides the infrastructure for building secure financial contracts using two core parts of their technology - a framework for building and deploying the derivatives and also an oracle known as the Data Verification Mechanism (DVM) to enforce them.

Unlike Synthetix, these financial contracts are designed to be “priceless”, meaning they do not require an on-chain price feed to be properly valued. Instead, the contracts will rely on proper collateralization of the contracts’ counterparties. This is incentivized by rewarding users who identify falsely collateralized positions. To further verify improper collateralization, the contracts may employ the use of the DVM.

The DVM is an oracle made to respond to price requests from these contracts and is only used to resolve disputes regarding liquidations and settlement of the contracts. The price requests are derived from UMA token holders, who vote on the most accurate value at a specific time. Token holders commit and reveal their votes over a multi-day process.

Once votes are revealed, the price or value with the most votes is relayed back to the financial contract. Collateral is then distributed to the token holders based on this voted figure. The DVM is built in such a way that there is an economic guarantee regarding the validity of the price requested.

Basically, this means that malicious actors are disincentivized from behaving badly. This is accomplished by ensuring that the cost of corrupting the DVM, taken as the total cost of more than half of the UMA voting tokens, is always greater than the profit from corruption or the total collateral available within

the contracts. This inequality is maintained by adjusting the price of the tokens through additional fees on the contracts.

Besides the DVM, there are five other important actors in the UMA ecosystem as well. These include the:

Token Sponsors - Individuals who deposit collateral in a smart contract to mint synthetic tokens. They are responsible for maintaining their own collateralization ratio to prevent liquidation.

Liquidators - Network of monitors that are incentivized to check if a position is properly collateralized through off-chain price feeds. There is a 2-hour delay for disputers to verify the accuracy of the liquidation before it is finalized.

Disputers - Disputers are incentivized users that monitor contracts. They reference their off-chain price feed to validate a liquidation event. If it is invalid, the DVM will be called into action.

DVM - The oracle will aim to resolve the dispute by proposing a vote on the most accurate price of the asset at a given timestamp.

Token holders - UMA token holders collectively vote on the price of an asset at a specific time. Token holders will reference off-chain data to provide information to the DVM. The DVM will then tally the votes and report the most agreed-upon price on-chain.

If the disputer is correct, the disputer and the affected token sponsor are rewarded. If the liquidator is correct, the liquidator will be rewarded, while the disputer would be punished, and the token sponsor loses their funds due to a finalized liquidation.

Some of the products that have been built using UMA's framework include synthetic "yield dollars", which are tokens that approach a specific value upon maturity, as well as uGAS tokens, which can be used to speculate on Ethereum gas prices. UMA has also introduced call options on popular DeFi tokens, such as Sushi and Balancer, as well as their own UMA KPI options.

These KPI options track the TVL of UMA and its price performance, determining the amount of UMA that the option holder can redeem upon redemption.

Now that you know a little more about UMA and how it works, let's compare these two synthetic asset platforms.

Comparison between Synthetic and UMA

Factors	Synthetix	UMA
Synthetic Token	Synths	uTokens
Underlying Assets	Fiat, cryptocurrency, commodities, equity indexes, equities	Virtually anything
Oracle	Chainlink	Data Verification Mechanism
Collateralization Ratio	500%	Depends on token minted
Ability to Stake	Yes	Yes

Both Synthetix and UMA offer over-collateralized synthetic assets, which could serve as proxies for a wide range of asset classes. On Synthetix, the value of Synths and the collateralization ratio are dictated by an on-chain price feed.

On UMA, the settlement of synthetic contracts is maintained by incentivizing network actors to behave. With a much more flexible collateralization requirement, which is based on the Global Collateralization Ratio for all token sponsors, we can see that UMA could be a more capital-efficient platform. However, Synthetix has an advantage with options, as it supports over 50 Synths in its catalog.

Moreover, Synthetix has built an ecosystem around Synths, with dHEDGE and Curve Finance's cross-assets swaps being some of the main drivers of demand. dHEDGE, a decentralized asset management platform, allows

users to invest sUSD into different investment portfolios. The chosen portfolio managers would then swap sUSD for other synthetic assets as part of their strategy. On the other hand, Curve Finance's cross-asset swaps make use of Synthetix as a bridge, allowing traders to swap up to 8 figures worth of assets, with zero to little slippage.

In terms of liquidity, it remains to be seen if UMA can compete without having a dedicated platform for trading these assets. Its exchange volumes are also low, recording only around \$20 million in daily trading volume. That said, synthetic assets still have a long way to go before we see mass retail and institutional adoption.

Notable Mentions

-  **Mirror Protocol**
Available on both the Ethereum and Terra blockchains, Mirror Protocol issues synthetic assets, called mAssets, that mimic the price of real-world assets such as stocks and indices. Some of the offerings include mAMZN and mQQQ.
-  **DEUS Finance**
A DeFi protocol that lets users source data from oracles and tokenize them as tradeable dAssets. dAssets assets are pegged 1:1 to their real-life counterparts using price oracle data.

Associated Risks

When dealing with decentralized derivatives platforms, it is important to note that leveraged trading and derivatives usage is a highly risky endeavor. Maintaining a healthy collateral ratio and keeping an eye on the liquidation price of your positions are essential to safely navigating this particular section of DeFi.

Since synthetic assets mostly rely on oracles as the primary source of price information, false data may lead to unwanted consequences. Additionally, since synthetic assets are mainly minted by depositing collateral, lack of

liquidity for these assets may occur, leading to drastically skewed pricing compared to real-world assets.

In the case of options, do ensure that you can exercise in-the-money positions in a timely manner as some platforms do not offer auto-exercise functions. As open interest continues to build up, one should take note of large option expiration periods, as volatility tends to increase.

Conclusion

Decentralized derivatives and synthetic assets offer regular users the opportunity to participate in previously inaccessible markets or markets that did not exist. These products have been simplified for the user's convenience and are no longer reserved for the elites. Users can also now participate in these derivatives markets without an intermediary.

This particular subset of DeFi is still relatively in its infancy, with a Total Value Locked of less than 10% of the entire space as of 1 April 2021. Indeed, bootstrapping liquidity is a problem faced by almost every up-and-coming DeFi project, and derivatives are no exception. Even with incentives, the very nature of these products with inherently higher volatility may seem to outweigh the rewards for participants to underwrite them.

Although protocols such as Charm and Perpetual Protocol can operate with minimal liquidity, it is still a long way to go before they can compete with big centralized exchanges that can offer larger volumes and higher leverage of up to 125x.

Recommended Readings

Decentralized Perpetuals

1. Documentation and Frequently Asked Questions about Perpetual Protocol
<https://docs.perp.fi/>
2. Research, Insights, and Announcements from dYdX
<https://integral.dydx.exchange>

3. The latest news from dYdX
<https://dydx.exchange/blog>

Decentralized Options

1. Articles and announcements from Hegic
<https://medium.com/hegic>
2. Hegic Protocol - On-chain Options Trading Protocol
<https://defipulse.com/blog/hegic-protocol-on-chain-options-trading-protocol/>
3. Opyn Review
<https://defirate.com/opyn/>
4. Beginner's Guide to Options: Opyn V2
<https://medium.com/opyn/a-beginners-guide-to-defi-options-opyn-v2-4d64f91acc84>

Synthetic Assets

1. Documentation & System Overview of Synthetix
<https://docs.synthetix.io/>
2. The latest news and announcements about Synthetix
<https://blog.synthetix.io/>
3. What is Synthetix? Everything you need to know about one of the leading DeFi protocols
<https://medium.com/coinmonks/what-is-synthetix-everything-you-need-to-know-about-one-of-the-leading-defi-protocols-bc19bdd2949c>
4. UMA Documentation
<https://docs.umaproject.org/>
5. List of Projects using UMA
<https://umaproject.org/projects.html>
6. UMA: Universal Market Access. Interview with Allison Lu
<https://defiprime.com/uma>

CHAPTER 8: DECENTRALIZED INSURANCE

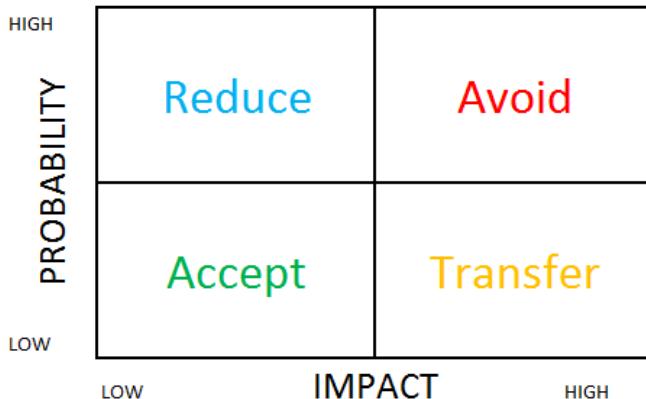
As DeFi projects launch and existing projects continue to innovate rapidly, we see an increasing number of hacks and exploits taking place, resulting in significant losses.

DeFi adoption will inevitably stall if the ecosystem only welcomes high-risk takers. Having a robust insurance system in place is a critical measure in reducing the risks users take on when interacting with DeFi applications, thus attracting more users to this space.

What is Insurance?

Insurance is a big industry, with total premiums underwritten globally reaching \$6.3 trillion⁴² in 2019. The world is inherently chaotic, and there is always the risk of accidents. Below is a simple risk management framework to show what we should do with different kinds of risks.

⁴² “World Insurance Marketplace | III - Insurance Information Institute.” <https://www.iii.org/publications/insurance-handbook/economic-and-financial-data/world-insurance-marketplace>. Accessed 6 May. 2021.



In this risk management framework, risks with high impact but with low frequency, such as natural catastrophes and terminal illnesses, should be transferred out. Insurance is created to deal with these types of risks.

How does Insurance work?

Insurance operates based on two main assumptions:

1. Law of Large Numbers

The loss event covered by insurance must be independent. If the event is repeated frequently enough, the outcome will converge to the expected value.

2. Risk Pooling

The loss event has the features of being low frequency and high impact. As such, insurance premiums paid by a large group of people subsidize the losses of several big claims.

Essentially, insurance is a tool to pool capital and socialize large losses so that the participants will not experience financial ruin with a single catastrophic event.

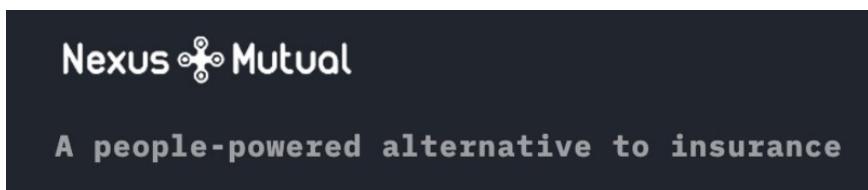
Does Crypto need Insurance?

Insurance empowers individuals to take risks by socializing the cost of any catastrophic events. It is an important risk management tool to encourage more people beyond the current niche user base to participate. The DeFi industry requires insurance products so that institutional players with significant capital to deploy are convinced that it is safe to participate in DeFi.

DeFi Insurance Protocols

We will be looking at three DeFi insurance protocols in detail below - Nexus Mutual, Armor Protocol, and Cover Protocol.

Nexus Mutual



Nexus Mutual is the largest DeFi insurance protocol in the crypto market by a wide margin. As of 1 May 2021, it has a Total Value Locked (TVL) of \$450 million compared to \$7 million by the second-largest DeFi insurance protocol, Cover Protocol. Nexus Mutual was founded by Hugh Karp, a former CFO of Munich Re in the U.K.

Nexus Mutual was registered as a mutual in the U.K. Unlike companies that follow a shareholders model, a mutual is governed by its members and only members are allowed to do business with the mutual. It's akin to a company run solely by the members for its members.

Type of Covers

Nexus Mutual offers two types of covers:

1. Protocol Covers

Cover DeFi protocols that custody users' funds as these smart contracts may experience hacks due to smart contract bugs. Protocol Covers were previously known as Smart Contract Covers and were upgraded on 26 April 2021 to include:

- Economic design failure
- Severe oracle failure
- Governance attacks
- Protection for assets on Layer-2
- Protection for non-Ethereum smart contracts
- Protection for protocols across multiple chains

Nexus Mutual offers covers for major DeFi protocols such as Uniswap, MakerDAO, Aave, Synthetix, and Yearn Finance.

2. Custody Covers

Cover the risks of funds getting hacked or when the withdrawal is halted. Nexus Mutual offers covers for centralized exchanges such as Binance, Coinbase, Kraken, Gemini, and centralized lending services such as BlockFi, Nexo, and Celcius.

In total, users can buy covers for 72 different smart contract protocols, centralized exchanges, lending services, and custodians.

Cover Purchase

To buy insurance from Nexus Mutual, users will first have to register as mutual members by going through the Know Your Customer (KYC) process. There is a one-time membership fee of 0.002 ETH. Once approved, users can then purchase cover using ETH or DAI.

Nexus Mutual will convert the payment into NXM, the protocol's token representing the right to the mutual's capital. 90% of the NXM is burned as the cover cost. 10% of the NXM will remain in the user's wallet. It will be used as a deposit when submitting a claim and will be refunded if there are no claims.

Claim Assessment

Users can submit a claim anytime during the Cover Period or up to 35 days after the Cover Period ends. As each claim submission requires users to lock 5% of the premium, users are therefore allowed to submit at most two claims for each policy.

Unlike traditional insurers, the claim result is decided through members voting - members have complete discretion on whether a claim is valid. Members can stake their NXM to participate as a claim assessor, subjected to a seven-day lock-up period.

When the vote is aligned with the result, 20% of the policy's premium will be shared proportionately with these members. However, when the vote is not aligned with the result, members will not receive any rewards, and the locking period will be extended by another seven days.

To be eligible for a valid claim, users will have to prove that they have lost the fund:

- Protocol Covers - lose at least 20% of funds
- Custody Covers - lose at least 10% of funds

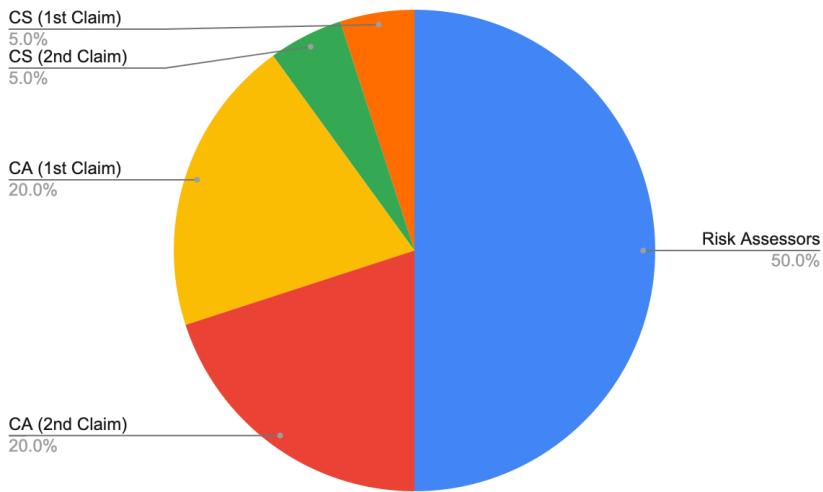
Risk Assessment

The pricing of the insurance is decided by the amount of capital staked on a particular protocol. Users can stake NXM on the protocols to become risk assessors - the more NXM staked on the protocol, the lower the cover price will be.

As of April 2021, 50,000 NXM is required to reach the lowest base pricing of 2%. A surplus margin, which is set at 30%, is devised to meet costs and create a surplus for the mutual. Factoring this in, the lowest possible cover cost is 2.6%.

The risk assessor bears the loss when there is a claim. For taking on this risk, 50% of the policy's premium is shared with the risk assessors.

Below is a pie chart showing where the premium flows to:



Claim Submission (CS): Fees paid by the user to submit a claim

Claim Assessor (CA): Fees earned by Claim Assessor if there is a claim submission

If no claims are submitted when the policy expires, 10% of the premium will be refunded to the cover buyer, while 40% of the premium will go to the capital pool.

Risk assessors are allowed to stake 15 times the capital available to maximize capital efficiency. For example, if a risk assessor has 100 NXM, he/she can stake 1,500 NXM across multiple protocols, with a maximum stake on any one protocol capped at 100 NXM.

The assumption here is that it will be very unlikely to have multiple protocols hacked at the same time. This practice aligns with how traditional insurance operates, based on the law of large numbers and risk pooling.

If the claim amount is larger than the capital staked by the risk assessors, the mutual's capital pool will pay the remaining amount.

To ensure there will always be enough capital to pay for claims, the mutual needs to have capital above the Minimum Capital Requirement (MCR).

Usually, MCR is calculated based on the risk of the covers sold. But due to the lack of claim data, the mutual follows manual parameters decided by the team.

Token Economics

NXM token economics is a big factor in attracting and retaining capital. It uses a bonding curve to determine NXM's token price. The formula is as follow:

$$\text{Price} = A + \left(\frac{\text{MCR}_{\text{ETH}}}{C} \right) \times \text{MCR\%}^4$$

$$A = 0.01028$$

$$C = 5,800,000$$

MCR (ETH) = Minimum Capital Required

MCR% = Available Capital / MCR (ETH)

MCR% is a key factor in determining NXM's token price as it has a power of four in the price formula. When people buy NXM through the bonding curve, available capital will increase, causing MCR% to grow, leading to an exponential increase in NXM's price.

The key thing to note here is that the bonding curve's withdrawal will be halted when MCR% is lower than 100%. This is to ensure there is enough capital to pay claims.

Wrapped NXM (wNXM)

In July 2020, the community members of Nexus Mutual released wrapped NXM (wNXM) as a way for investors to have exposure to NXM without going through the KYC process. When the withdrawal of NXM is halted (MCR% goes below 100%), users can wrap their NXM into wNXM and sell it through secondary markets such as Uniswap and Binance.

wNXM has many shortcomings as it cannot be used in risk assessment, claim assessment, and governance voting. The launch of the Armor protocol helps to solve the above issues by converting wNXM into arNXM.

Further details can be found below in the Armor Protocol's section.

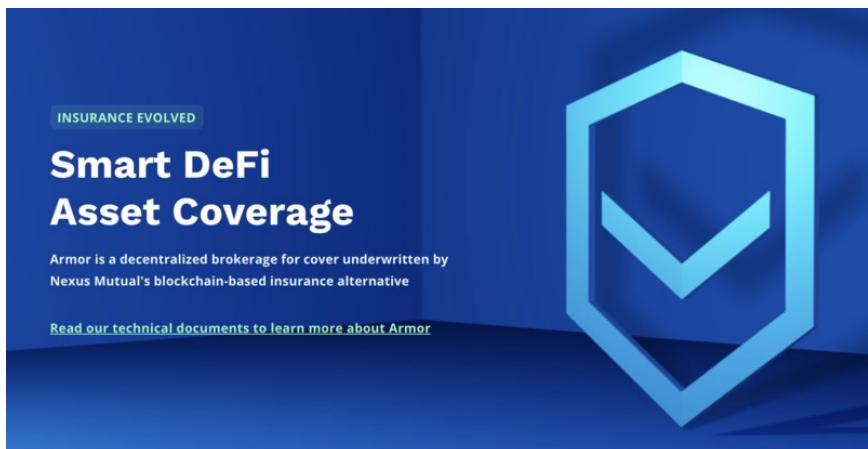
Protocol Revenue

NXM token differs from other governance tokens because a formula controls the token price. So, if the mutual is earning a profit, it will help increase the capital available and increase the price of NXM.

There are three sources of profit:

- Premiums collected - Claims paid - Expenses
- 2.5% spread when users sell NXM from the bonding curve
- Investment earnings from the capital pool

Armor Protocol



Armor makes investing in DeFi as safe as possible with crypto-native, dynamic smart coverage aggregation. As a decentralized smart brokerage, Armor's innovations provide on-demand, real-time coverage and non-custodial security solutions for user assets.

Armor protocol has four main products: arNXM, arNFT, arCORE, and arSHIELD.

arNXM

Nexus Mutual created Wrapped NXM (wNXM) to allow investors to have exposure to NXM without doing KYC. However, as more wNXM were created, less NXM became available for internal functions of the mutual such as staking, claim assessment, and governance voting.

Armor created arNXM to solve this issue by allowing investors to participate in Nexus Mutual's operations without doing KYC. To get arNXM, users can stake wNXM in Armor. Armor unwraps the wNXM, and the NXM token is then subsequently staked on Nexus Mutual. By staking on Nexus Mutual, stakers signal that the smart contracts are safe, opening up more insurance covers for sale.

arNXM can also be referred to as a wNXM vault. Users who deposit wNXM into the vault can expect to receive a higher amount of wNXM in the future.

arNFT

arNFT is the tokenized form of insurance coverage purchased on Nexus Mutual. arNFTs allow users to buy insurance cover without having to do KYC. Since these insurance covers are tokenized, users can now transfer them to other users or sell them on the secondary market. These tokenized covers also allow for further DeFi composability.

arNFTs can be minted for all Nexus Mutual's covers.

arCORE

arCORE is a pay-as-you-go insurance product. Using a streamed payment system, Armor tracks the exact amount of user funds as they dynamically move across various protocols and charges by the second. Underlying arCORE are pooled arNFTs that are broken down and sold at a premium. arCORE allows for much more innovative product design and showcases the composability nature of the DeFi ecosystem.

arCORE's products are charged at a higher premium to compensate arNFT stakers for taking the risk of not fully selling out the cover. As of April 2021, the multiplier is 161.8%, meaning the price would be 61.8% higher instead of purchasing directly from Nexus Mutual.

For the additional premium, 90% is given back to arNFT stakers, and 10% is charged by Armor as an admin fee. At a 1.618 premium multiplier and 90% share of revenue, utilization would have to be greater than 69% for this to be profitable for arNFT stakers. If the covers sold are less than 69% of those staked in the pool, then the stakers will have to foot the cover costs themselves.

arSHIELD

arSHIELD is an insured storage vault for Liquidity Providers (LP) tokens where insurance premiums are automatically deducted from the LP fees earned. arSHIELD essentially creates insured LP tokens where users do not have to pay upfront payments.

arSHIELD only covers the protocol risk of the liquidity pools. For example, insured Uniswap LP tokens only cover the risks of Uniswap's smart contract getting compromised, but not the risks of the underlying assets getting hacked (e.g., a hack of underlying asset protocol).

As such, arSHIELD is just a repackaged version of arCore.

Claim

After a user files a claim, a review process will be triggered and submitted to Nexus Mutual for consideration. Armor token holders will also participate in Nexus Mutual's process for claim approvals and payouts. If a payout is confirmed, the amount will be sent to Armor's payout treasury before being distributed to the affected users.

Protocol Revenue

Armor's focus is on building an ecosystem of interoperable protocols and products to secure and scale mass adoption of DeFi both with institutions and individuals.

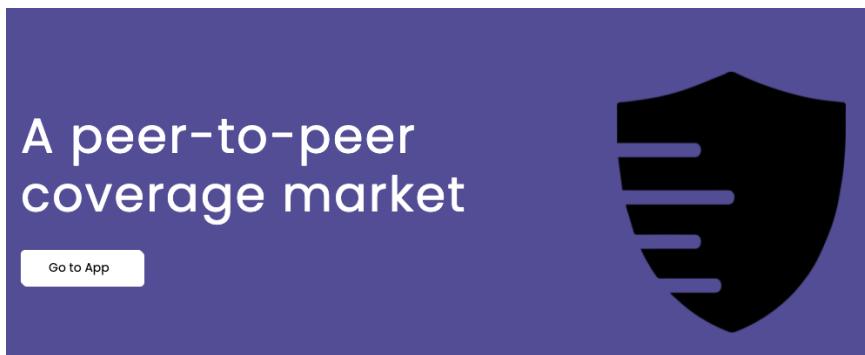
Below is the profit-sharing fees table updated as of February 2021:

Product	Stakers' Share	Treasury Share
arNXM	90%	10%
arCore	90%	10%
arNFT	0%	100%

Source: <https://armorfii.gitbook.io/armor/products/arcore/model-constants>

One thing to note is that 10% of the premium for every cover bought from Nexus Mutual is reserved for claim purposes. As the claim fee is 5% of the premium, every user can claim twice with the same policy. If there are no claims at the end of the policy period, the 10% premium will be refunded. This is the source of arNFT's profit.

Cover Protocol



Cover Protocol was incubated by Yearn Finance, starting as Safe Protocol that offers yInsure. But due to some infighting between the founder, Alan, and a prominent community member, Azeem Ahmed, the project was canceled. Azeem took over the yInsure product and released Armor Protocol, while Alan went on to release Cover Protocol.

Yearn Finance announced a merger with Cover Protocol to insure all of its yVaults with Cover Protocol. However, Yearn Finance ended the partnership on 5 March 2021.^{43,44}

Type of Covers

Cover Protocol only offers Smart Contract Covers.

Let's go through an example of how covers are sold. Market makers can deposit 1 DAI, and they will be able to mint one NOCLAIM token and one CLAIM token. Both tokens represent only the risk of a single protocol. The tokens are only valid under a fixed timeframe, such as half a year.

Two scenarios can happen after half-year:

- If there are no valid claim events, NOCLAIM token holders can claim 1 DAI, while CLAIM tokens will have zero value.
- If there is a valid claim event, CLAIM token holders can claim 1 DAI while NOCLAIM token will have zero value.

This is akin to a prediction market where users can bet whether the protocols will get hacked within a fixed timeframe.

Cover Protocol introduced partial claims, so the payout for CLAIM token holders when there is a valid claim event will be decided by the Claim Validity Committee (CVC).

Cover Purchase

Users can buy cover from Cover Protocol's web page with just one Ethereum transaction, without the need to register or do any Know Your Customer (KYC) process.

⁴³ "Yearn & Cover merger. Yearn and Cover Protocol join ... - Medium." 28 Nov. 2020, <https://medium.com/iearn/yearn-cover-merger-651142828c45>. Accessed 6 May. 2021.

⁴⁴ (2021, March 5). yearn.finance on Twitter: "We have decided to end the previously Retrieved May 23, 2021, from <https://twitter.com/iearnfinance/status/1367796331507552258>

Claim Assessment

There are two options for users to file for a claim:

1. Regular claim: A regular claim costs 10 DAI. COVER token holders will first vote on the validity of the claim. After being validated, it will move to the Claim Validity Committee (CVC) for a final decision.
2. Force claim: A force claim costs 500 DAI. The claim is sent to the CVC directly for a decision.

The CVC consists of external smart contract auditors. Cover Protocol will refund the claim filing cost if the claim is approved.

Risk Assessment

When users buy cover, flash loans are utilized to reduce the gas cost and steps required by the user. In this process, CLAIM and NOCLAIM tokens are minted with borrowed DAI. The NOCLAIM tokens are then sold to a Balancer pool for DAI.

Coupled with the premium paid by the user, the DAI is then used to pay back the flash loan, and users will only receive the CLAIM tokens. The reverse will happen when users sell the CLAIM token back to Cover Protocol.

There are a few benefits under this system:

- Cover cost is expected to reduce as there is only one Balancer pool to conduct yield farming programs. With the right incentives, market makers will buy more NOCLAIM tokens to yield farm or earn trading fees, pushing up the price of NOCLAIM tokens. As such, the price of the CLAIM token will reduce as $\text{CLAIM} = 1 - \text{NOCLAIM}$.
- Market Maker is expected to earn more fees as every cover purchase involved selling NOCLAIM tokens into the Balancer pool. Unlike

- the previous system, market makers only need to provide liquidity for one pool rather than two.
- Cover Protocol is expected to receive higher platform revenue as every purchase involves the CLAIM/NOCLAIM token minting with a 0.1% fee during redemption.

The cover price is decided by the supply and demand of the Balancer pool.

Protocol Revenue

0.1% fees will be charged on CLAIM and NOCLAIM token redemption. COVER token holders have the right to vote on how to use the treasury. As of April 2021, the staking of COVER tokens to earn dividends is being discussed, but details are not finalized.

Comparison between Nexus Mutual and Cover Protocol

	Nexus Mutual	Cover Protocol
Token Model	Mutual	Shareholder
Product	Insurance	Prediction Market
Risk Pooling	Yes	No
Capital Efficiency	High	Low
Counterparties covered	72	33
Claims	Voted by members	Voted by auditors
KYC	Not Required (Armor)	Not Required
Proof of Loss	Required	Not Required
Loss Covered	Full	Partial
Total Value Locked	\$450 million	\$7 million

As of April 2021, Nexus Mutual has a huge lead in the insurance market with seemingly no competitors in sight. But there is plenty of room for competitors to catch up as the insurance penetration rate in DeFi is very low,

with roughly only 2% of the total DeFi TVL.⁴⁵ In a field where innovations spring up each day, the title of insurance leader is always up for grabs.

Cover Protocol has been innovating rapidly, even throughout the Safe Protocol fiasco.⁴⁶ Even though the product has yet to gain significant traction, zero to one innovation is never easy. We have to remember that the Cover Protocol has been operational for less than a year (as of April 2021).

Capital Efficiency

Nexus Mutual allows capital providers to have 15 times leverage on the capital they stake. This translates into higher premium income for the stakers. Capital providers do have to take on more risks, but this approach is aligned with how the traditional insurance providers spread risk across multiple distinct products with different risk profiles.

In the meantime, capital providers for the Cover Protocol could not leverage their capital as every pool is isolated. As a result, Cover Protocol's covers are more expensive than those from Nexus Mutual due to less capital efficiency. For example, buying cover for Origin Dollar would cost 12.91% annually on Cover Protocol, while it only cost 2.6% on Nexus Mutual. There are plans to bundle different risks together in Cover Protocol Version 2, but details are still scarce.

We can calculate capital efficiency quantitatively by dividing the active cover amount over the capital pool. Nexus Mutual has a capital efficiency ratio as high as 200%, while Cover Protocol, by design, will always be less than 100%.

Covers Available

Cover Protocol only has coverage for 22 protocols, while Nexus Mutual has coverage for 72 counterparties. Nexus Mutual offers more flexibility in cover terms where users can decide to start the cover on any day and have a coverage period up until one year.

⁴⁵ (n.d.). Nexus Mutual Tracker. Retrieved May 23, 2021, from <https://nexustracker.io/>

⁴⁶ “Insurance Mining Hits a Speed Bump with SAFE Drama - DeFi Rate.” 16 Sep. 2020, <https://defirate.com/safe-insurance-mining/>. Accessed 10 May. 2021.

Cover Protocol only offers fixed-term insurance where the end date has been decided beforehand. For example, for a particular series, the insurance term is valid until the end of May. Regardless of when the user buys the cover, the cover will end in May. So as time goes by, CLAIM token will converge to \$0 while NOCLAIM token will converge to \$1.

Users can find more comprehensive offerings from Nexus Mutual as it has coverage for most of the main DeFi protocols. Even then, many covers are sold out due to the lack of stakers. The launch of Armor Protocol did help to alleviate the issue by attracting more wNXM into arNXM that allows NXM to be staked. As a result, more covers are available.

Cover Protocol can be seen as competing on long-tail insurance because projects can list much faster and do not have to go through cumbersome risk assessments. This is because every risk is isolated and contained within a single pool, unlike NXM, where a claim from any single protocol can eat into the capital pool. However, bootstrapping coverage for lesser-known projects is not an easy task. Other than being constrained by limited capacity, the insurance cost is often too expensive.

Claim Payout Ratio

Yearn Finance suffered an \$11 million hack in February 2021.⁴⁷ Even though Yearn Finance decided to cover the loss through their fund, insurance protocols have decided to pay out the claims to showcase that their product does work as intended.

Nexus Mutual has accepted 14 claims, amounting to a claim payout of \$2,410,499 (1,351 ETH and 129,660 DAI).⁴⁸ This resulted in a 9.57% loss to the NXM stakers who staked on Yearn Finance. The losses were fully paid if the claimants can show that they have indeed lost at least 20% of their fund.

⁴⁷ “Yearn.Finance puts expanded treasury to use by repaying victims of” 9 Feb. 2021, <https://cointelegraph.com/news/yearn-finance-puts-expanded-treasury-to-use-by-repaying-victims-of-11m-hack>. Accessed 6 May. 2021.

⁴⁸ “Paying claims for the Yearn hack | Nexus Mutual - Medium.” 24 Feb. 2021, <https://medium.com/nexus-mutual/paying-claims-for-the-yearn-hack-693bcfc5cd57>. Accessed 6 May. 2021.

Meanwhile, Cover Protocol decided to only have a payout percentage of 36% due to the loss being only 36% of the vault affected. If users held 1,000 CLAIM tokens, they received only \$360. Because there was only \$409,000 worth of CLAIM tokens available for Yearn Finance, market makers only lost \$147,240.

Cover buyers should realize that buying insurance from Cover Protocol does not guarantee a full payout in the event of a loss. The way the claim payout is decided is more similar to a prediction market.

Associated Risks

Claim payouts are highly dependent on the agreements set between insurance providers and the buyers. There are always nuances in interpreting agreements, especially in high-stakes scenarios that involve large claims.

Each insurance protocol will have its way of deciding what to pay, which may not necessarily be fair to all buyers. Buyers should be aware of the current limitation offered by the insurance products.

Being a capital provider for insurance protocols is a complicated operation, and users should have a complete understanding of the risks before deciding to get involved. Stakers can incur losses if the probability of claims is higher than expected.

Notable Mentions

-  **Unslashed Finance**
As of April 2021, Unslashed Finance is in private beta mode. Unslashed Finance offers bucket-style risk pooling for capital providers. The first product, named Spartan Bucket, covers 24 different risks covering counterparties such as custodians, wallets, exchanges, smart contracts, validators, and oracles.

Lido Finance purchased \$200 million worth of cover⁴⁹ from Unslashed Finance for its stETH (ETH 2.0 staking) to cover the risk of slashing penalties. Slashing refers to penalties exerted towards the Proof of Stake (PoS) network's validator when the validators fail to maintain the network consistently.

-  **Nsure Network**
Nsure Network raised a \$1.4 million seed fund from Mechanism Capital, Caballeros Capital, 3Commas, AU21, Signal Ventures, and Genblock back in September 2020.

Nsure Network is a marketplace to trade risk. It relies on the staking of NSURE tokens to signal the riskiness of a protocol and uses it to price cover. As of April 2021, they are running an underwriting program in Ethereum's Kovan testnet to assess how the pricing will work in mainnet. Participants will receive NSURE tokens as a reward.

-  **InsurAce**
InsurAce has raised \$3 million⁵⁰ from VCs such as Alameda Research, DeFiance Capital, ParaFi Capital, Maple Leaf Capital, Wang Qiao, and Kerman Kohli. It aims to become the first portfolio-based insurance protocol, offering both investment and insurance products to improve capital efficiency.

With InsurAce, users do not have to buy several covers if they are exposed to different protocols while doing yield farming, as it offers a portfolio-based cover covering all the protocols involved in the said investment strategy. It also claims to adopt an actuarial-based pricing model rather than relying on staking or market to price the cover.

⁴⁹ "Lido and Unslashed Finance Partner to Cover ETH ... - Lido.fi Blog." 23 Feb. 2021, <https://blog.lido.fi/lido-unslashed-finance-partner-to-insure-ethereum-staking-service/>. Accessed 6 May. 2021.

⁵⁰ "\$\$3M Raised during InsurAce Strategic Round | by ... - Medium." 25 Feb. 2021, <https://medium.com/insurace/3m-raised-during-insurace-strategic-round-ca94a7296dac>. Accessed 6 May. 2021.

As of April 2021, InsurAce has yet to announce its launch date. Due to the lack of claim history, it remains to be seen if InsurAce's portfolio-based insurance protocol and pricing model will work in the DeFi space.

Some derivative protocols also offer interesting insurance products such as:

-  Hakka Finance's 3F Mutual - covers the de-pegging risk of DAI.
-  Opium Finance - covers the de-pegging risk of USDT.

The adoption of these insurance products offered by the derivative protocols so far has been lackluster.

Unlike other decentralized exchanges and lending/borrowing protocols, insurance protocols seem to receive less attention. Besides being a more capital-intensive operation, the awareness of buying protection is not that prevalent in the crypto field. We may see more users getting onboard to use insurance with more insurance protocols slated to launch in 2021 and beyond.

Conclusion

The insurance market is still underexplored. According to the active cover amount of Nexus Mutual, only around 2% of the DeFi's Total Value Locked is covered. Derivatives products such as Credit Default Swap and options may dilute the need to buy insurance.

However, the construction of those products is usually more capital intensive than the risk pooling method of insurance, leading to more expensive covers. Plus, derivatives are rightfully more costly as they have exposure to price risk.

There is also the possibility that high-risk-takers and retail users dominate the current DeFi market; they may not have a strong emphasis on risk management and therefore do not consider the need for insurance. The

insurance market will gain more traction when the crypto space matures and has more involvement with institutional capital.

The underlying business of Nexus Mutual is humming well, with the active cover amount increasing ten times from \$68 million in January 2021 to \$730 million in February 2021.

Armor Protocol's launch has been a huge boon to Nexus Mutual, cementing Nexus Mutual's lead in the DeFi insurance market. As a wNXM vault, arNXM is intended to replace wNXM. It has attracted so much wNXM that arNXM now contributes 47% of the total NXM staked. This has helped to open up more covers for purchase. arNFTs meanwhile have contributed approximately 70% of the total active cover.

Cover Protocol has been innovating rapidly with new products such as Credit Default Swap, but their business growth has been rather slow. Cover Protocol offers fewer product offerings and has less flexibility in cover terms. But it allows projects to list faster and can offer coverage with relatively less capital.

Recommended Readings

1. Why DeFi insurance needs an Ethereum native claims arbitrator
<https://blog.kleros.io/why-defi-insurance-needs-an-ethereum-native-claims-arbitrator/>
2. Why insurance is needed for DeFi and what it looks like
<https://cryptoslate.com/why-insurance-is-needed-for-defi-and-what-it-looks-like/>
3. Nexus Mutual is the most undervalued token in digital assets
<https://twitter.com/jdorman81/status/1376920737949184002?s=20>

PART THREE: EMERGING DEFI CATEGORIES

CHAPTER 9: DECENTRALIZED INDICES

One way to get cryptocurrency exposure in your investment portfolio without constantly monitoring individual currency's performance is by investing in passively managed portfolios such as decentralized indices. Decentralized indices work similarly to Exchange Traded Funds (ETF) in the legacy financial markets.

An ETF is a type of structured security that can track anything such as an index (e.g., S&P500), commodities (e.g., precious metals), or other assets. It can be purchased or sold on a stock exchange.

Historically, ETFs have a better return than actively managed strategies like mutual funds. In 2020, global ETFs had \$7.74 trillion worth of Assets Under Management (AUM), with volume reaching one-third of global equity trading volume.^{51,52,53}

⁵¹ "Worldwide ETF assets under management 2003-2020 - Statista." 18 Feb. 2021, <https://www.statista.com/statistics/224579/worldwide-etf-assets-under-management-since-1997/>. Accessed 9 Mar. 2021.

⁵² "Institutional investors | iShares - BlackRock." <https://www.ishares.com/us/resources/institutional-investors>. Accessed 9 Mar. 2021.

⁵³ "ETF assets reach \$7tn milestone | Financial Times." 9 Sep. 2020, <https://www.ft.com/content/e59346a7-b872-4402-8943-2d9bdc979b08>. Accessed 9 Mar. 2021.

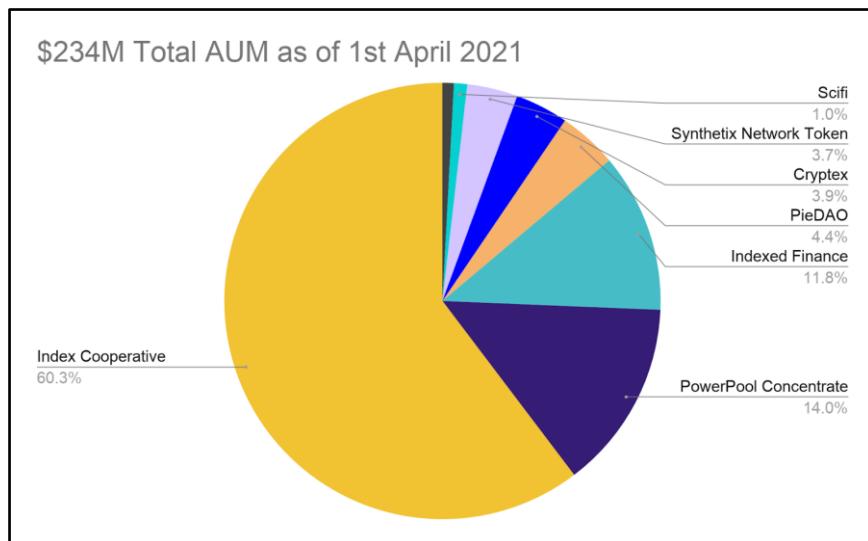
As of 1 April 2021, the decentralized indices sector is growing fast, with on-chain ETF amounting to approximately \$234 million in AUM.⁵⁴ It is not far-fetched to imagine this figure reaching trillions of dollars in the coming years.

In this chapter, we will be focusing on decentralized indices where you can diversify your portfolios without spending too much time and effort to research, manage, and allocate your investments.

Indices protocols refer to the asset managers, while index tokens refer to the indices' products (equivalent to ETFs). These index tokens represent your share of the index fund and entitle you to receive profits from the capital appreciation of the underlying assets. Additionally, there are governance tokens for some protocols, giving you voting rights in determining the direction of the indices protocols.

First, an overview of the decentralized indices market landscape:

DeFi ETF Landscape



Source: CoinGecko

⁵⁴ (n.d.). Top Index Coins by Market Capitalization - CoinGecko. Retrieved May 22, 2021, from <https://www.coingecko.com/en/categories/index-coin>

As of 1 April 2021, Index Cooperation has the largest market share with 60% of decentralized indices AUM. This is followed by PowerPool (14%) and Indexed Finance (12%).

Despite having over 20 DeFi index tokens in the market, the decentralized indices market is not as crowded and competitive as it seems.⁵⁵ The decentralized indices AUM comprises only 0.3% of the DeFi's Total Value Locked.

Let's take a look at the top 3 largest indices protocols - Index Cooperative, PowerPool Concentrated Voting Power, and Indexed Finance.

Index Cooperative (INDEX)



Index Cooperative, also known as Index Coop, is the oldest decentralized indices protocol. It was founded by Set Labs Inc., the same company that built Set Protocol.

Index Coop enables users to gain broad exposure to different protocols of varying themes across the cryptocurrency industry. Indices token holders can own, have exposure to, and can directly redeem the underlying assets that comprise the index.

Index Coop works with various methodologists - data providers that are accountable for the specific indices' strategy to launch its products.

As of April 2021, there are five indices available under Index Coop:

⁵⁵ “Top DeFi Index Coins by Market Capitalization - CoinGecko.” <https://www.coingecko.com/en/categories/defi-index>. Accessed 11 May. 2021.

- DeFiPulse Index (DPI)
- CoinShares Crypto Gold Index (CGI)
- ETH 2x Flexible Leverage Index (ETH2X-FLI)
- BTC 2x Flexible Leverage Index (BTC2X-FLI)
- Metaverse Index Token (MVI)

Indexed Finance (NDX)

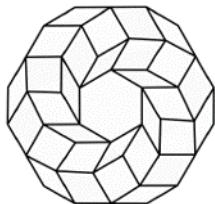


Indexed Finance is a protocol that focuses on portfolio management. Users can mint, swap or burn the indices token and the underlying assets, and the integrated Automated Market Maker (AMM) mechanism forked from Balancer rebalances its indices automatically. Indexed Finance has five team members, one of whom is anonymous.

As of April 2021, there are seven indices available under Indexed Finance:

- DEGEN Index (DEGEN)
- Cryptocurrency Top 10 Tokens Index (CC10)
- Oracle Top 5 Index (ORCL5)
- DEFI Top 5 Tokens Index (DEFI5)
- NFT Platform Index (NFTP)
- 484 Fund (ERROR)
- Future Of Finance Fund (FFF)

PowerPool Concentrated Voting Power (CVP)



PowerPool

PowerPool's indices are smart pools based on Balancer's Automated Market Maker (AMM) with additional functionality. Its main purpose is to pool governance tokens together for lending, borrowing, and executing meta-governance. Additionally, users can directly swap one governance token for another. An anonymous team runs PowerPool.

Currently, PowerPool has four indices:

- Power Index Pool Token (PIPT)
- Yearn Ecosystem Token Index (YETI)
- ASSY Index (ASSY)
- Yearn Lazy Ape (YLA)

Comparing the Protocol Indices

As a protocol investor, there are three primary metrics to look at:

1. Protocol fees
2. Protocol strategies
3. Fund weighting

Protocol Fees

Index Projects	Index Cooperative (INDEX)			Indexed Finance (NDX)					PowerPool Concentrated Voting Power (CVP)				
	DPI	CGI	FLI	DEFI5	CC10	DEGEN	ORCL5	NFTP	FFF	PIPT	ASSY	YETI	YLA
Index Funds	-	-	0.10%	-	-	-	-	-	-	0.10%	0.10%	0.10%	0.10%
Entry fee (mint)	-	-	-	-	-	-	-	-	-	0.10%	0.10%	0.10%	0.10%
Swap fee*	-	-	-	2.00%	2.00%	2.00%	2.00%	2.00%	2.00%	0.20%	0.20%	0.20%	0.20%
Asset Manager Treasury	-	-	-	-	-	-	-	-	-	0.10%	0.10%	0.10%	0.10%
LP return	-	-	-	2.00%	2.00%	2.00%	2.00%	2.00%	2.00%	0.10%	0.10%	0.10%	0.10%
Management Fee**	0.95%	0.60%	1.95%	-	-	-	-	-	-	-	-	-	-
Asset Manager Treasury	0.65%	0.24%	1.17%	-	-	-	-	-	-	-	-	-	-
Methodologist	0.30%	0.36%	0.78%	-	-	-	-	-	-	-	-	-	-
Exit fees (Burn/Redeem)	-	-	0.10%	0.50%	0.50%	0.50%	0.50%	0.50%	0.50%	0.10%	0.10%	0.10%	0.10%

Source: CoinGecko, Index Cooperation, Indexed Finance, Powerpool, Tokensets. Taken as at 1st April 2021

* When the user swap one of the underlying assets from one to another.

** Annualized

Index Cooperative

The management fees of each index are split between Index Coop and the associated methodologist. The fees are as follows - DPI: 0.95%, CGI: 0.60%, FLI: 1.95%. There are no exit fees for DPI and CGI. Only FLI has an exit fee of 0.1%.

Indexed Finance

To cover impermanent loss, a 2% swap fee is charged and distributed to the LP holders in the form of the input token in any swap when you mint or burn an index token from or to one of its component assets. This 2% fee is not charged if you mint using all of the underlyings, or redeem back to all the underlying assets.

However, you will be charged a fixed 0.5% fee when burning any of the index token, which is distributed back to protocol users who have staked the native NDX governance token.

PowerPool Concentrated Voting Power

There are three types of fees: Entry, Swap, and Exit fees. If you were to mint an index token, you would be charged 0.1% as an entry fee. 0.2% swap fees apply to users who swap one governance token for another. The swap fee is then split evenly between the index fund liquidity providers and the treasury. If you exit the indices, there is an additional 0.1% fee.

From the fee comparison above, Index Coop would stand to earn the most revenue as it charges the highest fees. Indexed Finance only has one source

of revenue - an exit fee of 0.5%. Powerpool meanwhile has a more diversified income from minting, swapping, and exit fees.

As a fund investor, you are likely to be a long-term holder, thus fees matter.

In this case, the index token PIPT would be the cheapest option relative to DPI and DEFI5. With PIPT and DEFI5, there is no ongoing cost, unlike DPI at 0.95% per annum.

Protocol Strategies

It is important we understand each protocol's strategies to understand their vision and direction on their decentralized indices products.

Index Cooperation

Below is a summary of how the team onboards a product on their protocol:

1. New product ideas are proposed and discussed with their community members
2. Product application is submitted to the community in the governance forum
3. The first snapshot vote is taken to move forward with vetting review
4. The passed proposals are reviewed by Index Coop's team.
5. The second snapshot vote is taken on product release
6. Product launch

Index Coop has a stringent process and needs two stages of community voting for product approval. For example, it took roughly three months to launch the Flexible Leveraged Index (FLI) since the first snapshot vote.

Indexed Finance

Relative to Index Coop, Indexed Finance moves faster.

For example, ORCL5, the first index fund that was up for voting, took a total of 18 days to launch starting from the voting stage.^{56,57}

Powerpool Concentrated Voting Power

The latest index product by Powerpool was Yearn Lazy Ape, which was put up for governance voting on 17 January 2021. It was only launched almost three months later, on 3 March 2021.^{58,59}

Currently, Index Coop and Powerpool have four index products under them. Indexed Finance has seven indices.

Although Indexed Finance appears to be the fastest one at launching indices, Index Coop and PowerPool teams work with their methodologist to ensure their products are safe and consider all associated factors and risks.

Indexed Finance may start moving slower with the launch of their Sigma program. The Sigma program allows Indexed Finance to collaborate with external partners, which would require a longer time. For instance, the DEGEN index fund, a collaboration with Redphonecrypto was announced in late December 2020 and only went live three months later.

Fund Weighting

Metrics	Index Cooperative (INDEX)		Index Finance (NDX)				PowerPool Concentrated Voting Power (CVP)			
	DPI	CGI	DEFI5	CC10	ORCL5	DEGEN	PIPT	ASSY	YETI	YLA
Fund Weighting	Market Cap-Weighted	A bi-level approach, accounting for historical volatility	Sqrt of Market Cap-weighted	Equal-weighted Market Cap	Market Cap-weighted	Market Cap-Weighted	Adaptive weights proportional to vaults TVL			

In general, there are three major approaches in weighing tokens in decentralized index tokens:

⁵⁶ (n.d.). Oracle Top 5 Token Index proposal - Proposals - Indexed Finance. Retrieved May 31, 2021, from <https://forum.indexed.finance/t/oracle-top-5-token-index-proposal/89>

⁵⁷ (2021, February 19). Indexed Finance on Twitter: “The \$ORCL5 index is live! The initial Retrieved May 31, 2021, from <https://twitter.com/ndxfi/status/1362839864106840064>

⁵⁸ (n.d.). PowerPool - Accumulate governance power in Ethereum based Retrieved March 14, 2021, from <https://app.powerpool.finance/>

⁵⁹ (2021, January 16). Power Pool \$CVP on Twitter: “Proposal 18: Yearn Lazy Ape Index Retrieved March 14, 2021, from <https://twitter.com/powerpoolcvp/status/1350658954607509505>

I. **Market Cap Weighted (e.g., DPI)**

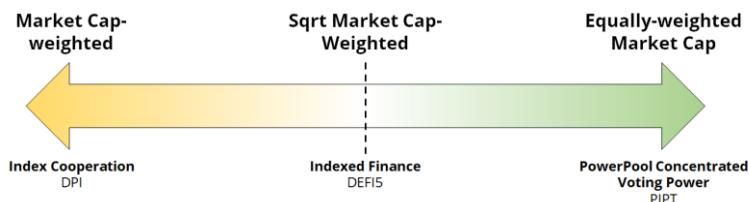
This method tracks each asset's market caps dynamically, where the allocation of each asset is proportional to their market capitalization relative to the other assets in the index. Indices using this method will be concentrated towards larger market capitalization coins relative to smaller ones, enabling the index to mimic the actual market performance closely.

II. **Square Root of Market Cap Weighted (e.g., DEFI5)**

All indices on Indexed Finance are based on the square root of the market capitalization relative to each underlying asset. This approach dampens the effect of market performance skewed towards larger market capitalizations coins.

III. **Equally-Weighted Market Cap Weighted (e.g., PIPT)**

This method sets asset allocation equally. For example, there are eight underlying assets in PIPT. Hence each of the assets is set to 12.5% weightage. An equal-weighted strategy is driven by price momentum and would favor smaller market capitalization coins. Smaller market capitalization coins are given the same importance as large market capitalization coins.



Associated Risks

Here are the three big risks when it comes to investing in these DeFi indices protocols and funds:

I. Code is Law

Although all the top-3 indices protocols have been audited, investors need to keep in mind that audited protocols are not hack-proof.^{60,61,62,63} Despite the audit, there have been numerous hacks happening in the crypto space, and more often than not, the funds are irrecoverable.

II. Mercenary Capital

Most indices protocols have liquidity mining programs to incentivize liquidity providers and bootstrap liquidity for the index tokens. However, much of these capitals are typically known as “mercenary capital”. These capital are purely searching for high returns and will exit as soon as another protocol with a higher yield appears. Consequently, once liquidity mining rewards dry up, it may result in mass withdrawals leading to a downward spiral for the indices protocol.

III. Systemic Risk

In DeFi, protocols can be stacked on top of each other like a money lego. However, composability can be a double-edged sword as it introduces systemic risk. For example, Yearn Lazy APE (YLA) by PowerPool has ten different risk exposures from its five underlying Yearn’s stablecoin vaults.

⁶⁰ (2020, September 4). Set Protocol Audit – OpenZeppelin blog. Retrieved March 17, 2021, from <https://blog.openzeppelin.com/set-protocol-audit/>

⁶¹ (n.d.). Security Audits - Power Pool. Retrieved March 17, 2021, from <https://docs.powerpool.finance/security/security-audits>

⁶² (n.d.). monoceros-alpha/audit-indexed-finance-2020-10 - GitHub. Retrieved March 17, 2021, from <https://github.com/monoceros-alpha/audit-indexed-finance-2020-10>

⁶³ (n.d.). maxsam4/indexed-finance-review - GitHub. Retrieved March 17, 2021, from <https://github.com/maxsam4/indexed-finance-review>

1. yvCurve-Compound (8.6%)	4. yUSD (27%)
2. yvCurve-3pool (36%)	5. yvCurve-BUSD (11.1%)
3. yCurve-GUSD (17.3%)	

These assets interact with ten different protocols and therefore have ten different risks. The ten protocols involved in Yearn's stablecoin vault are:

1. Yearn	6. Circle (USDC)
2. Curve	7. Gemini (GUSD)
3. Compound (cDai & cUSD)	8. Binance (BUSD)
4. Maker (DAI)	9. TrustToken (TUSD)
5. Tether (USDT)	10. PowerPool (YLA)

Notable Mentions

-  **BasketDAO - Interest Bearing DPI (BDPI)**
A product of the BasketDAO team, BDPI is the interest-bearing version of DPI. The difference is that the underlying assets are lent out to lending protocols such as Aave and Compound to earn a yield. As a result, the returns for holding this index fund is expected to be higher than DPI.
-  **Cryptex Finance - Total Crypto Market Cap (TCAP)**
Created by Cryptex Finance, TCAP allows you to have exposure to the entire crypto market. The team also runs Prysmatic Labs, one of the research teams behind ETH 2.0.

Conclusion

We are still very early in the decentralized indices sector, which we expect will grow rapidly in the coming months and years. As a long-term investor, we highly recommend that you look into a fund's past performance, check every chargeable fee, and consider the fund strategy utilized. It is also advisable to choose funds based on your risk appetite, factoring in the asset category you wish to have exposure in.

Recommended Readings

1. DeFi Indices Explained
<https://newsletter.banklesshq.com/p/the-best-defi-indices-for-your-crypto>
2. How to buy Indices with Argent
<https://www.argent.xyz/learn/how-to-buy-defi/>
3. Dashboard on DeFi Indices
<https://duneanalytics.com/0xBoxer/indices-products>

CHAPTER 10: DECENTRALIZED PREDICTION MARKETS

Prediction markets are markets created for participants to bet on the outcomes of future events. A great example of conventional prediction markets is sports betting platforms.

Decentralized prediction markets utilize blockchain technology to create prediction markets on just about anything. For example, prediction markets can be made based on when the Bitcoin price will exceed \$100,000 or who the next US president would be.

Proponents of decentralized prediction markets believe that centralized platforms put users at a disadvantage.⁶⁴ Standard practices include high transaction fees, delaying withdrawals, and freezing accounts. Moreover, most of today's traditional betting platforms are focused on sports betting, limiting the types of prediction markets available to the general public.

Decentralized prediction market protocols have been designed to empower users by allowing them to create their own markets.

⁶⁴ Beneš, N. (2018, April 6). *How manipulation-resistant are Prediction Markets?* Medium. <https://blog.gnosis.pm/how-manipulation-resistant-are-prediction-markets-710e14033d62>

How do Prediction Protocols work?

Unlike conventional prediction markets, prediction protocols are decentralized and have to rely on innovative methods to function. We can roughly break down a prediction's protocol process map into two primary sections:

1. Market-Making
2. Resolution

Market-Making

In market-making prediction protocols, there are two types of shares (outcome tokens) in a basic category-type market: YES (long) shares and NO (short) shares. Payout is determined based on whether an event occurs.

In a simple prediction market, a single YES share (which is often denominated as \$1) pays out \$1 if the event in question occurs and pays out \$0 if the event does not occur. Each NO share pays out \$1 if the event does not occur and \$0 if it does occur. Category-type markets utilize this basic principle, for example, 'Will BTC surpass \$100,000 by 31 December 2021?'

Another example of a question is: 'Who will be the US President in 2025? In this case, there may be more than two options, such as:

- A. Joe Biden
- B. Kamala Harris
- C. Trump

The above functions similarly as a basic category-type market, except three shares have been included to represent the three different answers instead of two. The price of shares is based on how much buyers are willing to pay and how much sellers are willing to accept. In other words, the system is an autonomous bookie whose rates (i.e. price) are determined by the market's weighing of probabilities.

On the other hand, markets with a range of answers and associated rewards will operate differently. Known as scalar markets, outcomes vary within defined parameters.

A good way to envision scalar markets is to think of them as possessing outcomes that set out to determine who is the most right/wrong as opposed to who is definitively right/wrong.

Let's use an example here with the following assumptions:

“What will the price of Bitcoin be on 10 November 2021?”

Precision = \$10k

Range = \$0 - \$200k

With this setup, a user may choose either \$10k, \$20k, \$30k, and so on.

Unlike YES/NO and Multiple Choice markets, the payout from scalar markets is distributed to all participants. Each payout is based on where the price falls within the range relative to the outcome. So if on the closing date, the price of BTC is \$198k, then the pot will be split between all purchasers. However, the closest answer to the \$200k strike price will receive the highest payoff amount in proportion to the size of their bets.

For a scalar market, price per share translates to a particular strike price in the underlying asset, or whatever is being predicted.

Resolution

Using the same example as before, how does one determine whether Bitcoin surpasses \$100,000 by 31 December 2021? Do I refer to Coinbase, or do I refer to the aggregate prices of all exchanges on CoinGecko? In practice, the market maker will specify the resolution source before its creation. So in this scenario, one may establish Coinbase as the resolution source.

The real issue is who provides this information, and how is that information validated? For price-based markets, public APIs can be drawn and extracted

from online sources. Oracles can also be used but may not cover all market types, such as “Will Vitalik Buterin marry before 2022?”

Considering the size and scope of prediction markets, purely relying on tech is impossible. Prediction protocols recognize this and rely on humans to ensure that the information is accurate.

However, how does one ensure that bad actors do not manipulate the market by supplying false information? Unlike conventional prediction markets, prediction protocols are decentralized and do not have the resources to monitor and regulate every market. To resolve this issue, prediction protocols have come up with different solutions. We will cover two examples in this chapter.

Prediction Market Protocols

Augur



Augur operates on the Ethereum network and uses a resolution model which incentivizes users to report information accurately through rewards and penalties.⁶⁵ After the market has closed, it will enter a reporting period where either the market creator or someone else (depending on who the market creator specified as the Designated Reporter) may supply the information to validate the results.

⁶⁵ Augur. *The Ultimate Guide to Decentralized Prediction Markets*. Augur. (n.d.).
<https://augur.net/blog/prediction-markets>.

During the reporting period, the Designated Reporter (DR) will have 24 hours to submit a report on the market's outcome. The catch is that the DR must stake the protocol's native token before reporting their findings.

There are two versions of the native tokens: REP and REPv2. REPv2 tokens only apply for Augur's v2 protocol update, whereas REP is migratable and redeemable for REPv2 tokens.

The key difference is that REP holders may not participate in a fork, should there be a substantial dispute over an outcome. However, we will collectively refer to them as REP tokens for simplicity's sake because they are functionally similar.

Whichever outcome the DR selects becomes the Tentative Winning Outcome (TOR). Once a DR submits the TOR, it is open to dispute. Contested results will open up the dispute period for one week, where anyone with REP can stake on the answer that they believe is correct. One round lasts for one week but can reach up to sixteen rounds.

If there is no dispute, a portion of the winnings is used to compensate the DR. This fee rate is variable and determined based on all REP tokens' total aggregate value in circulation.

If there is a dispute, users who staked on the winning outcome will get a share of the REP that was staked on the losing outcome. This is on top of the fee that reporters will receive. Users who staked on the losing outcome will not receive any fees and lose all their REP tokens.

Omen



Omen is a prediction protocol developed by DXdao and powered by the Gnosis protocol - it operates on Ethereum and the xDai sidechain.⁶⁶ Gnosis allows Omen users to engage with their token framework, an event-based asset class that comprises the building blocks of prediction markets.

Unlike Augur, Omen does not incentivize the community to report and resolve markets. Instead, they rely on a decentralized community-driven oracle known as Reality.eth which verifies real-world events for smart contracts.

Most markets will be resolved following through Reality.eth where the community members will decide on topics based on fees. Users on Reality.eth post bonds for their chosen outcomes, and they can be challenged by someone posting a new answer and doubling the bond. This may happen for several cycles until the posting stops, where the answer is determined by the last person posting their bond.

Once Reality.eth has internally completed their resolution obligations, they will supply the outcome to the respective Omen market. If an Omen user is unhappy with the findings, he/she may (through Reality.eth) appeal to an external arbitrator, Kleros.

Kleros randomly selects jurors from a juror pool and offers game-theory based incentives to ensure the anonymous voters reach consensus. Those who stake on the correct outcomes collect from those who staked on the incorrect outcomes (much like Augur).

⁶⁶ *Omen Prediction Markets*. Omen. (n.d.). <https://omen.eth.link/>.

Notably, DXdao, the self-governing organization behind Omen, may also decide to function as a competent arbiter in the future.

What are the other key differences between Augur and Omen?

As we have just discussed, both Augur and Omen have very different approaches to the resolution process. Augur addresses the oracle problem by creating an ecosystem of incentives and penalties which regulate the reliability of reporting information. Omen outsources their reporting needs to an external DAO (using similar principles as Augur's approach). In that sense, Augur is more self-sufficient.

In terms of liquidity, markets on Augur v2 use 0x's off-chain order - orders are collected off-chain, and settled on-chain. In contrast, Omen's Automated Market Makers operate similarly to DEXs like Uniswap, which creates large liquidity pools for token pairs.

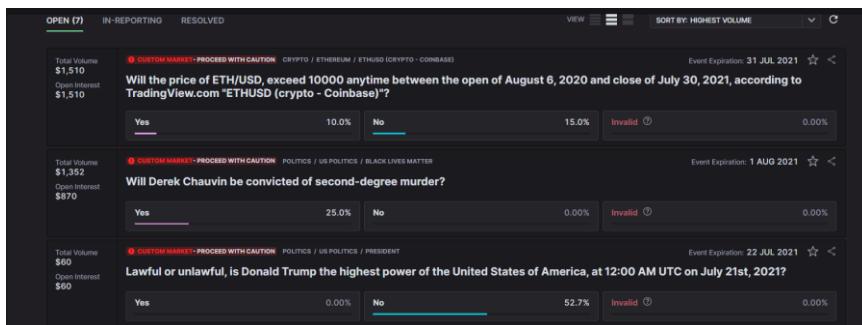
Although both outcome tokens use the ERC-1155 standard, Omen's tokens may be wrapped into the ERC-20 standard and accessed outside their network - this allows Omen to access DEXs and tap into larger liquidity pools. On the other hand, Augur's outcome tokens are confined to their protocol's internal liquidity pools.

For a concise comparison of Augur and Omen, you can refer to the table below:⁶⁷

⁶⁷ Gnosis. (2020, July 5). *Omen and the Next Generation of Prediction Markets*. Medium. <https://blog.gnosis.pm/omen-and-the-next-generation-of-prediction-markets-2e7a2dd604e>

	Augur	Omen
Information Validation System	Uses internal rewards and penalties to control	Outsourced to external DAO-type arbitrators
Liquidity	Traditional order book + 0x protocol for decentralized implementation	Automated Market Maker
Governance	Admin key that are burned means there are no further updates	DAO
Tokens	REP (REP + REPv2)	-
Outcome Tokens	ERC-1155	ERC-1155 but may be wrapped into ERC-20 tokens to access liquidity pools from other DEXs
Market Pairs	DAI	30 pairs
Supported Blockchains	Ethereum	Ethereum and xDai

We do not have access to either protocol's market data, but an arbitrary scan (conducted on 28 April 2021) of both protocol's websites will show that there is very little activity going on.



For Augur, there are only seven markets created. Only the top three have actual trading volume which collectively add up to just shy of \$3,000.

What will Joe Biden's Presidential Job Approval Rating be after 100 days in office? (Source: Gallup)

★ 54.00 % | 6 days remaining | 3.25K xDAI - Liquidity

What will the average USD price of CTX be on May 1st, 2021? (REF: most liquid of Uniswap, Sushiswap, or Swapr pools)

★ 18.50 USD | 5 days remaining | 2.19K xDAI - Liquidity

Will Compound Chain be launched and usable by the end of Q2 2021?

★ No (73.40%) | 2 months remaining | 1.63K xDAI - Liquidity

Will EIP-1559 be deployed on Mainnet before August 2021?

★ Yes (75.33%) | 3 months remaining | 1.13K xDAI - Liquidity

And as for Omen, there are only four markets but have a combined trading volume of roughly \$7,200 (not reflected in the image but is shown on another webpage).

Associated Risks

The biggest concern with prediction markets is the reliability of data. While there are multiple profit-based incentives to minimize data manipulation, irrational actors could try to jeopardize outcomes. Furthermore, challenged outcomes could lead to situations that are time-consuming and costly.

Notable Mentions

-  Polymarket

As of April 2021, Polymarket has a beta product that is live on Ethereum. The whitepaper for the protocol has yet to be published, but they appear to be a hybrid between centralized and decentralized

structures. The protocol seems far more popular than Augur or Omen - our quick scan on 28 April 2021 showed a booming market activity with over sixty prediction markets. The most popular market alone reached about \$2 million in trading volume.

Conclusion

Prediction markets is an interesting space because there are implications beyond betting - it allows users to hedge risks. Traditional derivatives allow buyers and sellers to hedge against particular outcomes by holding the right to trade a specific good in the future at a particular price. For example, a rice farmer may enter into a derivative contract to sell 1,000 kilograms of rice on 31 December 2021 for \$5,000 if he expects the price to be lower during that time.

Prediction markets can be based on anything and not just rice. Rather than hedge against the price of rice, the rice farmer may decide to hedge against the weather instead. In other words, prediction markets allow users to hedge against more specific risks.

Prediction protocols provide a platform for anyone to hedge against anything. Not only that, but prediction markets can also act as “de facto” polls. Participants in prediction protocols effectively share their opinions on different matters, which can be extrapolated for general insights on a wide range of topics.

The future for prediction protocols is exciting because their principles could be the foundation for real-life use cases. One could imagine using similar resolution systems to bring legal contracts on-chain, as the arbitration methods function very similarly to the juror-based justice system. However, in the near term, we expect more protocols to leverage the power of prediction protocols and utilize them to create innovative hedging instruments.

Recommended Readings

1. Prediction Market Basics
<https://augur.net/blog/prediction-markets>
2. Pricing in Prediction Markets
<https://medium.com/veil-blog/a-guide-to-augur-market-economics-16c66d956b6c>
3. How resistant Prediction Markets are to Market-Manipulation
<https://blog.gnosis.pm/how-manipulation-resistant-are-prediction-markets-710e14033d62>
4. Key differences between Augur and Omen
<https://blog.gnosis.pm/omen-and-the-next-generation-of-prediction-markets-2e7a2dd604e>

CHAPTER 11: DECENTRALIZED FIXED-INTEREST RATE PROTOCOLS

If we look at traditional finance, where most financial consumers reside, globalization has led to an increased demand for stable financial ecosystems. Indeed, it has been over 20 years since the European Parliament first acknowledged the need for more price stability in their working paper entitled “The Determination of Interest Rates”:

“The integration of the world’s financial markets is increasing the pressure of external factors in the determination of domestic monetary policies. In addition, though the approaches of the world’s major central banks towards the conduct of monetary policy differ in detail, there is broad agreement on fundamentals: the **pursuit of price stability and the stability of financial markets.**”⁶⁸

The key point here is integration. Analogous to how the crypto industry operates, the space has matured to a point where DeFi has become the industry standard for protocols. Often referred to as financial Legos, blockchain technology has allowed developers to integrate with other protocols and build innovative financial products. However, such progress does not change the fact that the crypto industry is unpredictable and highly volatile.

⁶⁸ Patterson, B., & Lygnerud , K. (n.d.). *The Determination of Interest Rates*.
https://www.europarl.europa.eu/workingpapers/econ/pdf/116_en.pdf.

Stable interest rates are an important facet of every financial ecosystem. Although there is an abundance of lending protocols and yield aggregators that offer interest rates to lenders in the crypto industry, relatively few of them offer fixed interest rates.

With the growing popularity of yield farming and the demand for more stable lending and borrowing rates, several DeFi protocols have attempted to address the ever-increasing demand for stable interest rates and become the hallmark for reliability. This has bred a new class of protocols known as Fixed-Interest Rate Protocols (FIRPs).

Compared to traditional finance, where fixed interest rates come in the form of fixed deposits (or bonds), FIRPs leverage their underlying tokenomics structure and offer different incentives to maintain their interest rates. At this point, the FIRP ecosystem can be broadly classified into two categories:

1. Lending/Borrowing
2. Yield Aggregators

Even under this umbrella classification, FIRPs come in many shapes and sizes. Each protocol has its method of “fixing” interest rates which leads to different use-cases. Some offer a “fixed interest rate” or a “fixed interest-earning ratio”. Moreover, some FIRPs do not offer fixed interest rates at all but rather create an environment that facilitates fixed-interest rates.

We will cover three examples in this chapter.

Overview of Fixed Interest Rates Protocols

Yield

YIELD

Yield is a decentralized lending system that offers fixed-rate lending and interest rate markets using a new kind of token called “fyTokens”. The current iteration (Version 1) includes fyTokens for the DAI stablecoin. Called “fyDai”, this new class of tokens enables fully collateralized fixed-rate borrowing and lending in DAI.⁶⁹

fyDai tokens are Ethereum-based tokens (ERC20) that may be redeemed for DAI after a predetermined maturity date. fyDai are analogous to zero-coupon or discount bonds.

To mint or sell fyDai, borrowers will have to put up ETH collateral that currently follows the same collateralization ratio as MakerDAO (150%). Lenders buy fyDai, which will typically be priced at a discount to DAI. The difference between the discounted value and 1 DAI (the maturity value) represents the lender’s lending rate or the borrower’s borrowing rate.

Although the value of fyDai reflects the borrowing/lending interest rate, it could also be traded in the market as a bond instrument on its own. This is possible because there are several “series” of fyDai, each with a different maturity date.

The system is tightly integrated and complementary to Maker. Maker users can “migrate” their DAI vaults into fyDai vaults, locking in a fixed interest rate for a period and converting back to a Maker vault after maturity.

Interest rates are determined by the market’s valuation of fyDai (with each series having its own maturity date). For lenders, a higher valuation of fyDai will lower the interest rates earned once it reaches maturity.

⁶⁹ *Introduction*. Introduction · GitBook. (n.d.). <https://docs.yield.is/>.

Conversely, a higher valuation of fyDai will lower the borrowing rate for borrowers as it would be sold off to purchase the respective stablecoin (e.g., DAI). This means that depending on the time of purchase of fyDai tokens, both borrowers and lenders can determine their borrowing/lending interest rate.

Example:

Assume that a borrower deposits 1.5 ETH as collateral and intends to borrow 900 DAI at an annual borrowing rate of 10%. Once executed, the borrower will receive 1000 fyDai with a value of 900 DAI - this will be automatically sold off in the marketplace by the protocol, and the borrower will receive 900 DAI. At the end of the one year maturity period, the borrower will have to repay 990 DAI if they wish to withdraw their collateral.

For lenders, assume that a lender lends 1000 Dai. In return, the lender receives 1000 fyDai which will accrue value as it approaches maturity. Initially, 1 fyDai received is worth 1 Dai, but after one year has lapsed, 1 fyDai will be worth 0.909 Dai. The lender may then redeem 1000 fyDai for 1100 Dai. This effectively puts the lending interest rate at 10%.

In practice, users can only select fyDai series that have been pre-programmed by Yield. This is similar to how regular bond instruments function, where there are different bond rates and maturity periods.

In Q1 2021, Yield has proposed integration with the MakerDao protocol that would permit MakerDao to be a fixed-rate Dai lender to Yield borrowers. The proposal has been accepted by MakerDao governance and is in the process of being integrated into the MakerDao protocol.

Yield is also expecting to launch version 2 of their protocol in Summer 2021. It will include new collateral types and permit borrowing of assets beyond Dai, like USDC and Tether.

Did you know?⁷⁰

In January 2021, an anonymous individual paid off his mortgage loan with a bank and is now paying down his refinanced home loan through DeFi protocol Notional Finance. Notional has similar functionalities as Yield as they also use a zero-coupon bond system via the introduction of a novel financial primitive called fCash.

Saffron.Finance



Saffron Finance is a decentralized yield aggregator for liquidity providers and was one of the first protocols to utilize a tranche-based system.⁷¹ Tranches are segments created from liquidity pools that are divided by risk, time to maturity, or other characteristics to be marketable to different investors.

With the various tranches, Saffron Finance users can select different portfolios based on their preferred risk appetite. More importantly, the Saffron Finance ecosystem creates an internal insurance system where investors in the higher-risk tranches insure investors in the lower-risk tranches.

Saffron Finance's native token, SFI, is primarily used as a utility token to access tranche A, the higher-earning tranche. However, SFI can also be staked to earn pool rewards and vote on protocol governance.

A tranche system allows one to divide up the earnings and create different earning rates for different pools. In Saffron Finance's case, the A tranche makes ten times the earnings of tranche AA. Tranche S offers a variable

⁷⁰ Fernau, O. (2021, February 2). *Engineer Becomes His Own Lender in First DeFi Mortgage*. The Defiant - DeFi News. <https://thedefiant.io/engineer-becomes-his-own-lender-in-first-defi-mortgage/>.

⁷¹ saffron.finance. Saffron. (n.d.). <https://app.saffron.finance/#docs>.

interest rate that balances the A and AA tranches; they are always in a perfect equilibrium to maintain the fixed-interest earning ratio of ten times between tranche A and tranche AA.

Example

If Tranche AA earns 100 DAI:

- 1) Tranche A will earn 1,000 DAI
- 2) Tranche S will earn DAI at a rate that ensures Tranche A pays off 10 times more than Tranche AA

If, however, there is a platform risk (e.g., black swan event), Tranche AA will get their deposited assets and earnings first - this is taken from Tranche A's principal and interest earnings.

Horizon Finance



Unlike conventional yield aggregators, Horizon allows users to create their own markets based on game-theory principles.⁷² Game theory envisions an environment where there are only rational actors. In such a hypothetical situation, buyers and sellers will make optimal decisions based on the available information.

Horizon allows users to submit their collateral into a liquidity pool, which is then lent to lending protocols such as Compound. To provide fixed interest rates to users, Horizon invites users to submit their sealed bids for fixed interest rates (acting as yield caps) or floating interest rates in each round.

⁷² *Whitepaper*. Horizon. (n.d.). <https://docs.horizon.finance/general/whitepaper>.

The bids are revealed after each round, thus creating an order book of bids. The protocol will rank the bids from the lowest interest rate to the highest interest rate. The lending protocol's variable earnings are then distributed from the lowest interest rate bids to the highest interest rate bids, with any excess income spilling over into the floating pool.

One notable feature is that all bids will be displayed on Horizon's website. The displayed bids allow users to actively compete and ascertain which interest rates are the most popular. On top of that, users can freely amend their bids, including switching to the floating rate. Horizon essentially doubles up as an interest prediction protocol.

Example

To illustrate this, let's say that the round for Pool X lasts from 1 May 2021 - 14 May 2021:

On 1st May,

- Participant A deposits 100,000 DAI and bids that he will earn a 20% interest rate.
- Participant B deposits 100,000 DAI and bids the floating rate.

On 7th May,

- Participant C deposits 100,000 DAI and bids that he will earn a 10% interest rate.

At this point, Participant A reconsiders his bid as Participant C submitted a much lower bid. If Pool X earns too little, he may not get anything at all.

On 13th May,

- Participant A amends his bid to a 5% interest rate.

After the round ends, let's say the 300,000 DAI in Pool X managed to earn an interest rate of 4% for a total of 461 DAI, therefore:

- Participant A fulfills his bid and gets 192 DAI, earning 5% interest rate.
- Participant C partially completes his bid and gets the remainder 269 DAI, which is a 7% interest rate. His original bid of 10%, if fully fulfilled, would have generated 383 DAI during the two weeks period if Pool X had earned sufficient interest.
- Participant B fails in his bid and does not get anything.

As you can see, there are a lot of mind games involved! Moreover, interest rates are not technically fixed. However, the system rewards users who can gauge the amount of interest they should earn from their bids. This incentivizes users to conform to a ‘safe’ bid if they are uncertain about the amount they could earn. Bidding too high or bidding the floating rate could result in lesser gains or none at all. Thus ‘safe’ bids effectively become the ‘de facto’ fixed interest rate over time.

Which FIRP should I use?

FIRPs cannot be lumped into a single basket and compared side-by-side. For one, lending protocols are very different from yield aggregators.

Before even looking at more profit-oriented metrics like the competitiveness of interest rates, we should be looking at the FIRP’s ability to maintain their “fixed interest rate”, which is effectively their functionalities. And if we were to breakdown how FIRPs operate, there are essentially three defining characteristics that revolve around their promise of fixed-interest rates:

I. What kind of promises are they making?

Different protocols make different promises. For example, Saffron Finance promises that if you participate in Tranche A, you will earn 10 times more than Tranche AA. Horizon does not even make any promises as to how much you would earn. Understanding the type of promise made allows users to decide which protocol offers their preferred product.

II. How do they intend to maintain that promise?

Each type of promise requires a different methodology. For example, Saffron Finance offers insurance to Tranche AA users by giving them the earnings first, in the event of a deficit. Understanding how each promise is maintained allows users to determine which protocol is more reliable.

III. How dependent are they on external agents to maintain that promise?

Developing protocol mechanisms that influence user behavior are essential to all FIRPs. For example, Yield requires relatively even ratios of lenders and borrowers to maintain fixed interest rates. Identifying such traits allows users to determine how exposed the protocol's promise is to factors outside their direct control.

If we consider these criteria, it is impossible to say which would be the best fit for you. Ultimately, it boils down to each individual's preferred risk appetite, the type of financial instrument required, and the belief in the underlying protocol's mechanisms. And perhaps more importantly, the industry is still nascent because many protocols are still getting established - they have yet to prove themselves, especially during difficult market conditions which threaten their ability to offer fixed-interest rates.

Associated Risks

One of the most significant risks is a FIRP's ability to provide fixed-interest rates. Most of these protocols rely on external agents or other users to actively participate in the protocol to drive market functionality.

If there is an inactive community or a disproportionate amount of user profiles and liquidity (e.g., more lenders than borrowers for Yield, or more participants in Tranche A than Tranche AA for Saffron Finance), FIRPs may not be able to back their fixed-interest rates.

Notable Mentions

-  **Notional**
Notional facilitates fixed-rate, fixed-term lending and borrowing of crypto-assets. Much like Yield Protocol, both protocols have very similar functionalities as Notional creates a zero-coupon bond system via the introduction of a novel financial primitive called fCash. There are, however, some key differences. In particular, Notional has a different Automated Market Maker and different collateral options.
-  **BarnBridge**
BarnBridge leverages a tranche system (similarly to Saffron Finance) for yield-based products. However, BarnBridge also has another product (SMART Alpha) that offers exposure to market prices through tranches of volatility derivatives.
-  **88mph**
88mph is a yield aggregator which offers fixed-interest rates. They are able to maintain their rates through the introduction of floating-rate bonds and a unique tokenomics structure that helps influence market behavior.
-  **Pendle**
Pendle is an upcoming protocol that allows users to tokenize future yield, which can then be sold for upfront cash. Essentially, Pendle will calculate your expected yield, effectively locking in your interest rates.

Conclusion

FIRPs are a new suite of protocols that are bound to become a staple in the DeFi scene. We highlighted three examples because they are innovative and able to showcase DeFi's potential when combined with traditional fixed-income instruments.

There are a lot of exciting developments in this space that offer unique products and services. We already have protocols combining price prediction and yield aggregation; imagine if a bank offered competitive betting services on fixed deposit yields? We have not even discussed protocols that tokenize future yields, which essentially allow anyone to create their own bonds and sell them off for upfront cash.

As this area develops further, we expect more institutional interest in FIRP products. Fixed-income instruments have always been commonplace in traditional finance. However, as aggregate debt levels and inflation continue to rise, and the value of the US dollar continues to fall, FIRPs may offer more reliable yields.

Recommended Readings

1. Report on Tranche-based Lending in DeFi
<https://consensys.net/blog/codefi/how-tranche-lending-will-bring-fixed-interest-rates-to-defi/>
2. Fixed-Interest Rate Protocol Highlights
<https://messari.io/article/fixed-income-protocols-the-next-wave-of-defi-innovation>
3. Why Fixed-Interest Rates are Important
<https://medium.com/notional-finance/why-fixed-rates-matter-1b03991275d6>

CHAPTER 12: DECENTRALIZED YIELD AGGREGATORS

Crypto gave birth to the activity of yield farming, where users can earn yields just by allocating capital in DeFi protocols. Many crypto natives have since become yield farmers, searching for farms that offer the most attractive yields.

Due to the sheer number of new yield farms being released each day, no individual can be aware of every opportunity. With sky-high returns, the opportunity cost of missing out on new yield farms is increasingly high.

Yield aggregators are born to serve the need of automating users' investment strategies, sparing them the trouble of monitoring the market for the best yield farms. Below we are going to look into several decentralized yield aggregator protocols.

Yield Aggregators Protocols

Yearn Finance



ABOUT
yearn.finance
yearn.finance defi made simple

Yearn Finance started as a passion project by Andre Cronje to automate capital-switching between lending platforms to search for the best yield offered by DeFi lending platforms. This is needed as most DeFi lending platforms provide floating rather than fixed rates. Funds are automatically shifted between dYdX, Aave, and Compound as interest rates change between these protocols.

The service includes major USD stablecoins such as DAI, USDT, USDC, and TUSD. For example, if a user deposits DAI into the Yearn Finance, the user will receive a yDAI token in return, a yield-bearing DAI token.

Later on, Yearn Finance collaborated with Curve Finance to release a yield-bearing USD token pool named yUSD. Curve Finance is a decentralized exchange that focuses on trading between assets with roughly similar value, such as USD stablecoins. yUSD is a liquidity pool that includes four y-tokens: yDAI, yUSDT, yUSDC, and yTUSD.

Holding yUSD allows users to have five sources of yields:

1. Lending yield of DAI
2. Lending yield of USDT
3. Lending yield of USDC
4. Lending yield of TUSD
5. Swap fees earned by providing liquidity to Curve Finance

yUSD is thus promoted as a superior crypto USD stablecoin than just holding the underlying stablecoins.

Vaults

Yearn Finance debuted the vault feature after its token launch, igniting a frenzy on automated yield farming, and is considered the initiator of the category of yield farming aggregator. The vaults will help users to claim liquidity mining rewards and sell the protocol's native tokens for the underlying assets.

Vaults benefit users by socializing gas costs, automating the yield generation and rebalancing process, and automatically shifting capital as opportunities arise. Users also do not need to have proficient knowledge of the underlying protocols involved. Thus the vaults represent a passive investing strategy for users. It is akin to a crypto hedge fund, where the goal is to increase the number of assets that users deposited.

Besides simple yield farming, Yearn Finance also integrated various novel strategies to help increase the vaults' return. For example, it can use any assets as collateral to borrow stablecoins and recycle the stablecoins into a stablecoin vault. Any subsequent earnings are then used to buy back the asset.

Yearn version 2 was launched on 18 January 2021.⁷³ Version 2 vaults can employ multiple strategies per vault (up to 20 strategies simultaneously), unlike version 1 vaults that only employ one strategy per vault.

Strategies

As a yield aggregator, Yearn Finance has leveraged the composability feature of Ethereum to the maximum extent possible. Below, we will examine how Curve Finance's liquidity mining program plays a role in Yearn Finance's vault strategy.

Curve Finance is a decentralized exchange that focuses on stablecoins pairs. It utilizes a fairly complicated governance system - veCRV is used to measure the governance voting power, which users can get by locking their CRV tokens.

⁷³ (2021, January 18). Yearn Finance Launches v2 Vaults, YFI Token Jumps ... - BeInCrypto. Retrieved May 24, 2021, from <https://beincrypto.com/yearn-finance-v2-vaults-yfi-token/>

- 1 CRV locked for 4 years = 1 veCRV
- 1 CRV locked for 3 years = 0.75 veCRV
- 1 CRV locked for 2 years = 0.50 veCRV
- 1 CRV locked for 1 year = 0.25 veCRV

veCRV can be used to vote for enlisting new pairs and decide how much CRV yield farming rewards will be given to each pair. More importantly, veCRV is used to determine the boosted yield farming reward available for the liquidity providers.

Pool	Base APY ▼	Rewards APY
 y USD yDAI+yUSDC+yUSDT+yTUSD	22.91%	+8.68% → 21.69% CRV

By referring to the image above, yUSD is a pool of yield-bearing stablecoins. Users can deposit yUSD into Yearn Finance to obtain yCRV, where the CRV rewards will be harvested and sold to get more yUSD.

The Base Annual Percentage Yield (Base APY) refers to the swap fee earned by being a Liquidity Provider of the Curve pool. Rewards APY refers to the liquidity mining program rewards in the form of CRV tokens. With the use of veCRV, the base rewards of 8.68% can be scaled up to 21.69%, or 2.5 times from the base. In total, the expected return is roughly 31.59% to 44.60%.

By depositing your USD stablecoins into Yearn Finance, you will benefit from the maximum 2.5 times boosted yield farming rewards instead of having to lock your CRV to gain the boost.

Yearn Finance Partnerships

From 24 November 2020 until 3 December 2020, Yearn Finance announced a series of partnerships (dubbed Mergers and Acquisitions) of several protocols, essentially forming an alliance revolving around YFI.⁷⁴

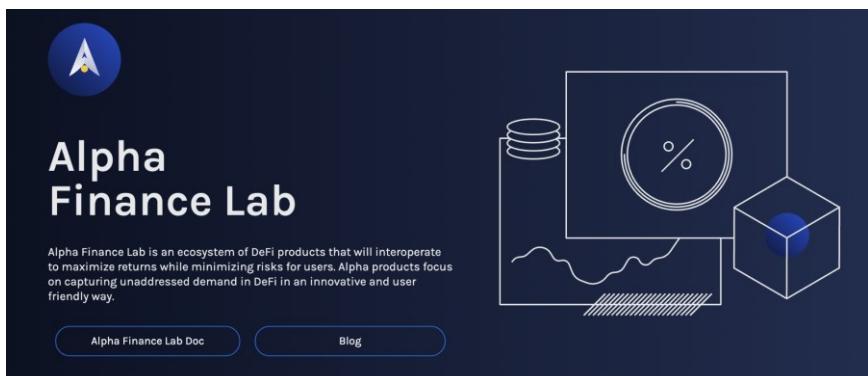
⁷⁴ (2020, December 4). Defi Lego connects as Yearn Finance announces five mergers in a Retrieved May 24, 2021, from <https://bravenewcoin.com/insights/defi-lego-connects-as-yearn-finance-announces-five-mergers-in-a-week>

- SushiSwap joined as its Automated Market Maker (AMM) arm
- Cover joined as its insurance arm
- CREAM joined as its lending arm
- Akropolis joined as its institutional service provider for vaults and upcoming lending products.
- Pickle joined as one of its strategists.

Yearn Finance has chosen to end the partnership with Cover Protocol on 5 March 2021.⁷⁵

Yearn Finance's version 2 also incentivizes contributions from the community by sharing a percentage of profits to community strategists. Yearn Finance has also established an affiliate program with other protocols that are willing to form synergistic relationships whereby the protocols stand to receive up to 50% of revenue generated. In other words, Yearn Finance has become a large ecosystem that offers a range of yield-farming products and services.

Alpha Finance



Alpha Finance introduced leveraged yield farming through their first product Alpha Homora, allowing users to use borrowed capital to increase their exposure in their yield farming activities. Essentially, it is acting as both a lending and yield aggregator protocol.

⁷⁵ (2021, March 5). yearn.finance on Twitter: “We have decided to end the previously Retrieved May 24, 2021, from <https://twitter.com/iearnfinance/status/1367796331507552258>

In Alpha Homora version 2, users can lend (to earn lending interest rate) and borrow many assets (to leverage their yield farming position), including ETH, DAI, USDT, and USDC, YFI, SNX, sUSD, DPI, UNI, SUSHI, LINK, and WBTC.

Example

Using the example of SUSHI/ETH as mentioned in [Chapter 2](#), rather than yield farming with only \$1,000 capital, with Alpha Homora, you can now choose to leverage two times your capital by borrowing \$1,000 worth of ETH.

By borrowing \$1,000, you will now participate in yield farming by providing \$1,000 worth of ETH and \$1,000 worth of SUSHI, totaling \$2,000. This strategy will only yield a profit when the swap fees and yield farming rewards are greater than the borrowing cost on Alpha Homora.

Also, note that since both ETH and SUSHI are available as borrowable assets, you can yield farm on leverage by borrowing both ETH and SUSHI to minimize swapping fees.

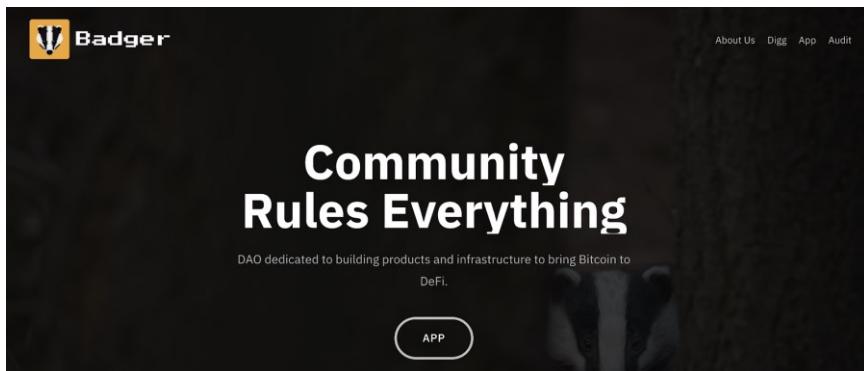
The borrowing cost on Alpha Homora is calculated at a variable rate, affected by supply and demand. If the borrowing cost spikes suddenly due to increased borrowing, then the leveraged position may incur a loss. Another risk is when the borrowed asset increased in price as compared to the yield farming position. Using the example above, if ETH increases rapidly in price while SUSHI drops in price, the leveraged position may experience a liquidation.

Besides earning higher returns, having a leverage position on the yield farms will also expose users to higher Impermanent Loss. The profit earned is highly influenced by the choice of asset borrowed to yield farm. For example, borrowing ETH vs. USD stablecoins will result in a completely different return profile. For more details about Impermanent Loss, do refer to [Chapter 5](#).

Alpha Homora V2 also supports Liquidity Provider (LP) tokens as collateral. For instance, users with liquidity providing position on ETH/SUSHI pool on Sushiswap will be able to deposit ETH/SUSHI LP token as collateral on

Alpha Homora V2 and borrow more ETH and SUSHI tokens to leverage yield farm.

Badger Finance



Badger DAO aims to create an ecosystem of DeFi products with the ultimate goal of bringing Bitcoin into Ethereum. It is the first DeFi project that chose to focus on Bitcoin as the main reserve asset rather than using Ethereum.

A Sett is a yield farming aggregator focused on tokenized BTC. Setts can be categorized into three main categories.

a) Tokenized BTC Vaults

- Inspired by Yearn Finance's vaults, initial products include Bitcoin vaults that farm CRV tokens such as SBTCCURVE, RENBTCCURVE, and TBTC/SBTCCURVE metapool.
- They also collaborate with Harvest protocol to farm CRV and FARM tokens with RENBTCCURVE deposited in Harvest itself.

b) LP vaults

- To attract more users, there is a Sett for WBTC/WETH that farms SUSHI rewards.
- Other than that, four Setts are created to bootstrap liquidity for BADGER and DIGG.
 - 1) WBTC/BADGER UNI LP

- 2) WBTC/DIGG UNI LP
- 3) WBTC/BADGER SUSHI LP
- 4) WBTC/DIGG SUSHI LP

c) Protocol Vaults

- Users can choose to avoid Impermanent Loss and tokenized BTC risks just by staking the native BADGER and DIGG tokens into bBADGER and bDIGG vaults, earning protocol fees and yield farming rewards.

Trivia: The word Sett is chosen as it refers to a badger's home.

Harvest Finance

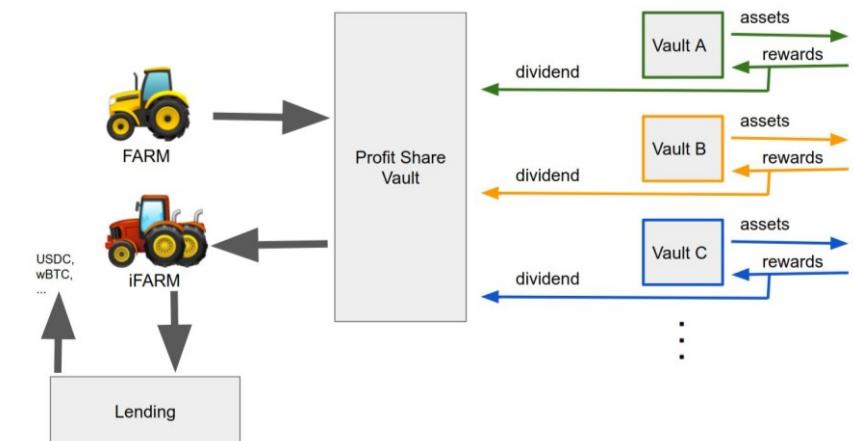


Starting as a Yearn Finance fork, Harvest Finance has since adopted a fast-mover strategy. It has been releasing new strategies faster than other yield aggregator protocols, even those deemed high risk.

As of April 2021, it supports an astounding 63 different farms in Ethereum alone, with categories covering stablecoins, SushiSwap, ETH 2.0, BTC, NFT, 1inch, algorithmic stablecoins, and mAssets by Mirror Protocol.

It has also recently expanded into Binance Smart Chain, offering farms on Ellipsis, Venus, Popsicle Finance, PancakeSwap, Goose Finance, and bDollar.

Harvest Finance team has released an interest-bearing FARM (iFARM) token where users can stake their FARM to earn the protocol fees.



Source: <https://mbroome02.medium.com/harvest-101-understanding-ifarm-and-its-potential-54d9cf305e5>

Comparison of Yield Aggregators

	Yearn Finance	Alpha Finance	Badger DAO	Harvest Finance
Management Fee	2%	-	-	-
Performance Fee	20%	-	20%	30%
Borrowing Fee	-	10%	-	-

One of the important factors to consider when deciding which yield aggregators to use is the fees charged. Yearn Finance follows the standard hedge fund model, where it charges a 2% management fee and 20% performance fee. Badger DAO and Harvest Finance charge only a performance fee of 20% and 30%, respectively. Alpha Finance's Alpha Homora v1 on Ethereum and BSC collect 10% of the interest charged based on the amount borrowed for leverage and collects 20% on Alpha Homora V2.

The fee structure suggests that users may have to pay the highest fees by investing with Yearn Finance - 2% of the amount invested is taken away annually regardless of whether the strategies deployed are earning a return or not.

Charging performance fees can be considered fairer as it just means a lower return for the users. Yearn Finance can charge a premium due to its industry-leading position, with a lot of its vaults being integrated with other protocols such as Alchemix, Powerpool, and Inverse Finance.

	Yearn Finance	Alpha Finance	Badger DAO	Harvest Finance
TVL (\$ Mil)	2,000	936	1,140	527
Market Cap (\$ Mil)	1,305	434	287	113
FDV (\$ Mil)	1,348	1,810	975	180
Market Cap/TVL	0.65	0.46	0.25	0.21
FDV/TVL	0.67	1.93	0.86	0.34

* Data taken as of 1 April 2021

Yearn Finance still maintains the lead in terms of Total Value Locked (TVL), while Harvest Finance seems to be the most undervalued among the Yield Aggregators. Meanwhile, Alpha Finance is the most overvalued, based on the ratio of Fully Diluted Valuation against Total Value Locked (FDV/TVL).

Associated Risks

Yield aggregators are exposed to a high risk of hacks due to their nature of seeking high yields from riskier protocols. Out of the four protocols, only Badger DAO has yet to be hacked (as of April 2021).

Integration with insurance protocols is still lackluster, which may be the biggest bottleneck to further growing the sector's Total Value Locked. With the launch of more insurance protocols, we may see the launch of insured yield aggregator products in the future.

Notable Mentions

-  **Pancake Bunny**
 Pancake Bunny is the biggest yield aggregator in the Binance Smart Chain ecosystem. It only provides farms that are based on PancakeSwap. The low gas fee on Binance Smart Chain allows for a

more frequent re-staking strategy, thereby compounded yield and resulting in higher APY. The offered farms consistently provide yields that are higher than 100%.

-  **AutoFarm**
AutoFarm is a cross-chain yield farming aggregator, supporting Binance Smart Chain and Huobi ECO Chain. Like Pancake Bunny, AutoFarm offers a higher frequency of compounding and thus higher APY for its farms. It is the second-largest yield aggregator in the Binance Smart Chain ecosystem.

Conclusion

Yield aggregators have a similar role to actively managed funds or hedge funds. Their job is to find the best investment opportunities and earn fees from them.

In DeFi, liquidity mining programs have given birth to a specialized way of earning returns. With DeFi composability being utilized in increasingly creative ways, we predict the strategies employed by yield aggregators will get more complicated.

Most yield farming programs only live for roughly three to four months and can be changed anytime by governance. Yield aggregators help users find high-yielding farms, but new farms usually have increased risk of being hacked. It is challenging to balance the search for high yields with risks.

There is also the worry that the high yield offered by the yield aggregators may not be sustainable. As of April 2021, the high yields are partly supported by the speculative market environment. For example, a high CRV token price translates into high yield farming rewards. No one knows for sure how yields may behave in a bear market, but it has a high chance of compressing to zero. That will not be a good sight for the yield aggregators.

Recommended Readings

1. Yearn Improvement Proposal (YIP) 56 - Buyback and Build
<https://gov.yearn.finance/t/yip-56-buyback-and-build/8929>
2. Yearn Improvement Proposal (YIP) 61: Governance 2.0
<https://gov.yearn.finance/t/yip-61-governance-2-0/10460>
3. Upcoming Alpha Homora V2 Relaunch! What Is Included?
<https://blog.alphafinance.io/upcoming-alpha-homora-v2-relaunch-what-is-included/>

PART FOUR: TECHNOLOGY UNDERPINNING DEFI

CHAPTER 13: ORACLES AND DATA AGGREGATORS

DeFi is powered by smart contracts. Sometimes, the inputs required to produce an output consist of real-world data not stored on the blockchain, such as weather conditions or traffic information. There is a need for protocols to bridge the gap by relaying off-chain data onto the blockchain for smart contracts to interact with the data.

Off-chain information is an integral part of DeFi and should always be valid and accurate. Having false data will completely misrepresent a particular project and cause major problems for DeFi. However, how do we ensure that the data provided is always accurate and can be trusted?

Some protocols aim to achieve this by transmitting and broadcasting data onto the blockchain without being manipulated or tampered with. This is usually done through a voting or consensus mechanism where validators agree on the most accurate data. Without oracles or data aggregators as the main “source of truth”, bad actors can make use of false information to take advantage of unsuspecting users.

In this chapter, we will take a closer look at some of the oracles and data aggregators available such as Chainlink, Band Protocol, Graph Protocol, and Covalent. We will see how these oracles and data aggregators bridge the gap between blockchains and real-world data.

Oracle Protocols

Oracles act as a bridge between off-chain data and the blockchain, or between protocols that do not have internal data feeds to reference on-chain data. These oracles seek to relay external information to the blockchain to be verified and executed upon by smart contracts or Dapps in the DeFi ecosystem.

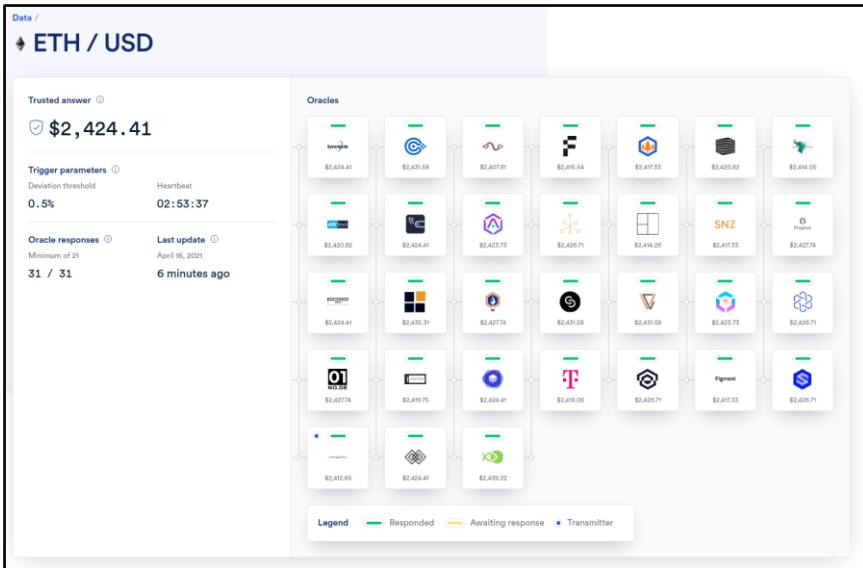
Chainlink



Chainlink is a framework and infrastructure for building decentralized oracle networks that securely connect smart contracts on any blockchain network to external data resources and off-chain computation. Each oracle network is secured by independent and Sybil-resistant node operators that fetch data from a multitude of off-chain data providers, aggregate the information into a single value, and deliver it on-chain to be executed upon by smart contracts.

One of the main functions of Chainlink is to deliver the most accurate asset prices through its price feeds, which can be integrated into blockchain protocols for specific use cases. For example, asset prices are very important when settling options and futures contracts upon maturity and when the assets are used as collateral for loans. Chainlink's services also include a 'Proof-of Reserve' reference feed for cross-chain tokens and a Verifiable Random Function for on-chain gaming applications.

In order to dive deeper into how Chainlink interacts with and processes data, we will look at some of the methods used by Chainlink oracles to bridge real-world data onto the blockchain.



The most common method that Chainlink oracles use to bring external data on-chain is the Decentralized Data Model, a continuously updated on-chain smart contract representing a specific piece of data (e.g., the price of ETH against BTC) that can be queried on-demand in a single transaction.



The Basic Request and Receive Model is another method where a user's smart contract requests data directly from one or multiple Chainlink nodes and the reported value is received in the next transaction. This model is used to fetch random values or more unique datasets. Chainlink nodes are paid in LINK tokens as a fee for their services in both of these oracle network models.

Anybody can become a Chainlink node operator and start providing data to the network. Chainlink's Price Feed networks are secured by nodes operated by a combination of traditional enterprises like Deutsche Telekom's T-

systems, data providers, and professional DevOps firms. Data providers who operate their own Chainlink node cryptographically sign their data directly at the source, providing smart contracts with greater security guarantees.

In April 2021, the Chainlink 2.0 whitepaper was released, introducing a new architecture for Decentralized Oracle Networks. Decentralized Oracle Networks functions similarly to Layer-2 solutions, significantly increasing the speed of data delivery and boosting security. Furthermore, Chainlink introduced its super-linear staking model, a crypto-economic security mechanism that incentivizes nodes to deliver accurate oracle reports and significantly increases the cost of attack for malicious actors.

That's a quick overview of Chainlink and decentralized oracle networks! You may have noticed: some of the top DeFi protocols introduced in this book, such as Synthetix and Aave, are also powered by Chainlink.

Band Protocol



Similar to Chainlink, Band Protocol is a cross-chain oracle platform that connects smart contracts with external data and APIs. Decentralized applications that have been integrated with Band Protocol receive data through Band Protocol's smart contract data points instead of directly from off-chain oracles.

A unique aspect of Band Protocol is that they utilize BandChain, a separate blockchain to handle and relay information that can handle thousands of transactions. Data can be sent to other blockchains via Cosmos' Inter-Blockchain Communication (IBC) protocol.

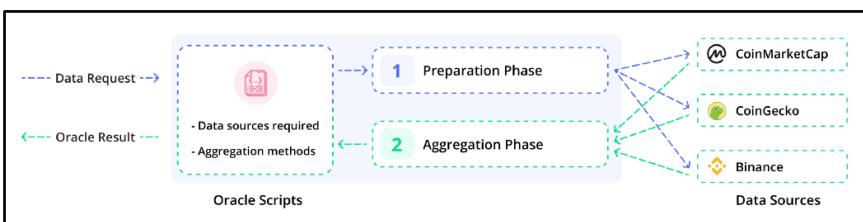
Data sourced from Band Protocol are curated and verified by the community, ensuring that they are reliable enough to be referenced by Dapp users and developers alike. These data sources can be aggregated from

various on-chain feeds and data aggregators using different statistical methods such as mean, median, or mode, as well as any additional data-cleansing methods such as normalization or time-weighted averages.

To retrieve data from Band Protocol, the requester will need to specify several parameters: the oracle script ID the requester wants to call, the parameters for the oracle script, and the number of validators required. Once the request has been made and verified on BandChain, the oracle script will begin the first phase of execution, known as the preparation phase, by emitting the data sources required to fulfill the request.

Validators will be chosen to handle the request based on a randomized stake-weighted algorithm. The validators will attempt to retrieve the requested data from all the specified sources and submit a raw data report, with the retrieved results, to the BandChain for confirmation.⁷⁶

Once the minimum number of validators have successfully submitted their reports, the BandChain will proceed with the second phase, also known as the aggregation phase. All collected reports are compiled into a single result stored on BandChain, which can be accessed and sent to other blockchains for future use.



The BandChain network relies on multiple participants, the most important being validators and delegators. The top 100 validators with the most staked BAND tokens are responsible for creating and confirming new blocks on the BandChain network. On the other hand, delegators can delegate their BAND tokens to any validators to earn block rewards.

⁷⁶ (2020, July 20). Understanding Band Oracle #2 — Requesting Data on BandChain Retrieved April 29, 2021, from <https://medium.com/bandprotocol/understanding-band-oracle-2-requesting-data-on-bandchain-b3fde67072a>

Whenever requested, validators on BandChain have the duty of fetching data from specified data providers. Validators are economically incentivized to provide accurate data since the provision of false data will result in the slashing or confiscation of staked BAND tokens. Subsequent counts of misinformation will result in lower trust scores and user count, further decreasing the value of the staked tokens and compounding the losses.

On top of that, the data request process along with the validator's execution is publicly available for all to see and verify, further mitigating the risk of transmitting false or tampered data. Validators have served approximately 4.3 million requests for data in the first six months since oracle functionality went live on BandChain's mainnet in October 2020.

To become a validator on BandChain, you will need to own some BAND tokens or have other users delegate BAND tokens to you. Validators are chosen at random using a stake-weighted algorithm based on their share of staked tokens. The larger the proportion, the higher the chances of getting selected.

Data Aggregators

If oracles bridge real-world data to the blockchain, then data aggregators help users to read it. These protocols compile blockchain data into a simplified format, making it easier for projects and individual users to create their analytics dashboard.

The Graph Protocol



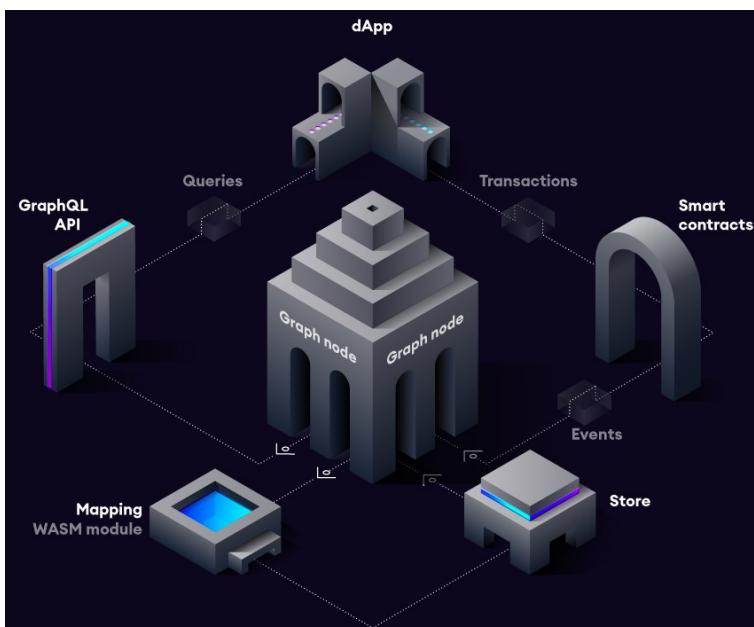
The Graph is a decentralized protocol for querying and receiving data from blockchains such as Ethereum, Polkadot, and Solana. Although it is simpler

to answer queries directly retrieved by reading a contract address on the blockchain, data with higher specificity and granularity is much more difficult to find. The Graph solves this problem by indexing blockchain data into “subgraphs”, or open APIs that can be queried using a standard GraphQL API.

The Graph works by well indexing diverse types of data based on the subgraph’s description, also known as the subgraph manifest. The manifests define the relevant smart contracts and the contract’s key events to focus on. It also figures out how to map the event data to the stored data in The Graph’s database.

Once a subgraph manifest is created, The Graph CLI stores it in the InterPlanetary File System (IPFS), a decentralized storage solution, and begins indexing information for that subgraph.

Here is a basic flow on how data is detected and stored by Graph Nodes:



1. Decentralized applications (Dapps) add data from a smart contract transaction to the blockchain.
2. The smart contract emits events while processing the transaction.
3. Graph Nodes continuously scan the blockchain for new blocks and data for subgraph manifests.
4. The Graph Nodes find the events and run the mapping handlers provided. The WASM module generates and updates the data stored within the nodes.
5. Dapps query Graph Nodes for indexed data using the GraphQL endpoint.
6. GraphQL queries are translated by the nodes before they are retrieved by the dApp.
7. The dApp displays this data via its user interface to be used as reference when issuing new transactions on the blockchain.

The Graph is used by many dApps in the DeFi and Web3 space, including Uniswap, Aave, Balancer, and Synthetix. It provides the proper infrastructure to service data-intensive protocols. As of Q1 2021, over 10,000 subgraphs have been deployed by approximately 16,000 developers and over 100 billion queries processed in less than a year.⁷⁷

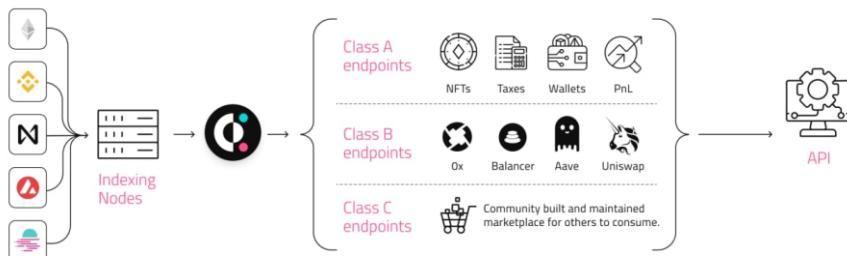
⁷⁷ (2021, February 24). The Graph's Indexing and Querying is now live on the Fantom Layer Retrieved March 4, 2021, from <https://thegraph.com/blog/graph-fantom>

Covalent



Covalent is a blockchain data provider that provides a unified API for accessing the seemingly endless rows of on-chain data. Detailed information for token balances and wallet activity from various blockchains can be retrieved using a single API, making it much easier for developers to create analytics dashboards or gather insights on blockchain activity.

Covalent's functions may seem quite similar to The Graph, but it differentiates itself in a few ways. For instance, the Graph requires a subset of data to be converted into a subgraph before it can be queried. Covalent, meanwhile, will index the blockchain in its entirety, resulting in larger quantities of granular data for users. Furthermore, Covalent has expanded beyond Ethereum. As of April 2021, Covalent supports data from four other blockchains, including Fantom, Binance Smart Chain, Polygon, and Avalanche.



Source: <https://www.covalenthq.com/blog/beginners-guide-to-covalent/>

To access the Covalent API, you will need to first obtain a free API key by creating an account on Covalent. There are two classes of Covalent API - Class A and Class B.

You may use Class A endpoints to retrieve blockchain data that are network-agnostic. In short, this is general data that applies to all networks, such as

balances, transactions, and token holders. On the other hand, you may use Class B endpoints to return values from specific protocols on a blockchain. For example, you may query data from Uniswap or PancakeSwap, both decentralized exchanges isolated on their respective networks.

The screenshot shows the Covalent API documentation for the endpoint `/v1/[chain_id]/address/{address}/balances_v2/`. The page includes a sidebar with navigation links for Overview, Authentication, Response Format, and Supported Networks. The main content area displays the request parameters:

- PATH PARAMETERS**
 - `chain_id`: string (Chain ID of the Blockchain being queried. Currently supports: [3] for Ethereum Mainnet, [137] for Polygon/Matic Mainnet, [8000] for Polygon/Matic Mumbai Testnet, [5] for Binance Smart Chain, [43114] for Avalanche C-Chain Mainnet, [41113] for Fuji C-Chain Testnet, and [258] for Fantom Opera Mainnet.)
 - `address`: string (Passing in an `ETH` resolves automatically.)
- QUERY-STRING PARAMETERS**
 - `alt`: boolean (Set to `true` to return ERC721 and ERC1155 assets. Defaults to `false`.)
 - `no-nft-fetch`: boolean (Set to `true` to skip fetching NFT metadata, which can result in faster responses. Defaults to `false`, and only applies when `alt=true`.)
 - `format`: string (If `format=csv`, return a flat CSV instead of JSON responses.)

A "TRY" button is located at the bottom right of the form.

Source: <https://www.covalenthq.com/docs/api/>

Notable mentions



DIA

DIA, short for Decentralized Information Asset, is an open-source oracle that provides data for many DeFi applications. They provide price feeds for popular DEXs such as Uniswap and Sushiswap on other chains such as Binance Smart Chain and Polygon.



API3

API3 provides information to projects via decentralized APIs or dAPI. The data feeds are overseen by a DAO which includes project partners and industry experts. They also allow dAPI users to obtain on-chain insurance in the event that the API does not work as intended.

Associated Risks

As more protocols start to rely on long-standing products that have been battle-tested in the harshest environments, it is also important to remember that these products may not be completely foolproof. If black swan events were to happen, the oracles might not be able to provide data efficiently, causing inaccurate decisions to be made.

In March 2020, otherwise known as Black Thursday, oracles such as Chainlink and MakerDAO's 'Medianizer' failed to update their price feeds quickly enough, resulting in grossly incorrect prices. MakerDAO's price failure kickstarted a chain of catastrophic events, leading to over \$8 million worth of ETH collateral lost for CDP owners.

Conclusion

Oracles and data aggregators form the backbone of many DeFi protocols. The need for extreme speed and accuracy in querying and relaying data is of utmost importance for future projects seeking to change the data provision game. Currently, Chainlink is dominating the space with over 400 integrations, including more than 200 DeFi projects.

However, the DeFi scene is constantly on the bleeding edge of innovation, with better oracle and data collection mechanisms coming into play on a regular basis. Ultimately, the goal of DeFi is to have an oracle service that is reliable, internally secure, and well-protected from negative externalities. Additionally, the robust indexing protocols that have been built and continuously improved will serve to bring more clarity and insight into the behavior of users on the blockchain, allowing projects to provide better products and services with greater product-market fit.

Recommended Readings

1. Documentation and Additional Information on Chainlink
<https://docs.chain.link/docs>
2. Chainlink Whitepaper
<https://link.smartcontract.com/whitepaper>
3. What Is Chainlink and Why Is It Important in the World of Cryptocurrency?
<https://finance.yahoo.com/news/chainlink-why-important-world-cryptocurrency-110020729.html>
4. Documentation on Band Protocol
<https://docs.bandchain.org/>
5. What is Band Protocol and How to Buy BAND
<https://decrypt.co/resources/what-is-band-protocol-defi-oracle>
6. Documentation on The Graph Protocol
<https://thegraph.com/docs/>
7. The Graph - Google on Blockchains?
<https://finematics.com/the-graph-explained/>
8. Documentation on Covalent API
<https://www.covalenthq.com/docs/api/#overview>
9. Here's a Quick Look at the Role of 'Covalent Blockchain Data API'
<https://newspi.site/heres-a-quick-look-at-the-role-of-covalent-blockchain-data-api-in-terms-of-data-gathering-e-hacking-news/>
10. Blog Post on Covalent Network Launch
<https://www.covalenthq.com/blog/covalent-network-blog/>

CHAPTER 14: MULTI-CHAIN PROTOCOLS & CROSS-CHAIN BRIDGES

Ethereum is undoubtedly the go-to home for many DeFi projects. However, the spikes in gas fees due to the high usage levels and the run-up of Ether to its All-Time High have served as a wake-up call for users and developers to explore other blockchains with lower transaction fees.

Many DeFi projects found a second home in other blockchain networks to service even more users and scale more efficiently. Although exchanges usually offer the ability for users to transition easily between various blockchain networks, it can still restrict the movement of funds.

From this dilemma, several projects such as Ren, THORChain, and Anyswap have grown to allow users to seamlessly connect and move funds between blockchains in a trustless manner. Centralized exchanges such as Binance have also tried to bridge Ethereum and other blockchains by introducing the Binance Bridge.

Protocols and Bridge Overview

Ren Project



Ren Project is a permissionless protocol that allows users to interact and transfer tokens between blockchains anonymously. Ren Protocol does this via RenBridge, an offering powered by the Ren Virtual Machine (RenVM).

RenBridge allows for the easy representation of cryptocurrencies on other networks. For example, Bitcoin can be represented on the Ethereum Network as an ERC-20 token by having it wrapped into renBTC. Using RenVM, the assets are converted based on the format of its destination network at a 1:1 ratio, ensuring that the wrapped versions are always fully backed by the underlying asset.

While you explore other blockchains with your newly wrapped tokens, RenVM acts as a decentralized custodian of your original assets, with the value of the bonded REN maintained at three times the total value of the locked assets via minting and burning. The Darknodes are continuously shuffled to maintain the smooth operation of RenVM, and additional levels of security such as the implementation of the RPZ MPC algorithm and algorithmically adjusted fees make it extremely challenging for hackers to attack. In the unlikely event that an attack happens, RenVM can restore the stolen funds.

RenVM works by running on a decentralized network of computers known as Darknodes, which help verify transactions and maintain the security of the Ren network. To operate a Darknode, users need to stake 100,000 REN tokens as collateral, which amounts to approximately \$103,000 (7 May 2021).

The Darknodes collect a portion of the trading fees from RenVM transactions in the form of wrapped assets.

The Ren Virtual Machine supports three types of cross-chain transactions - lock-and-mint, burn-and-release, and burn-and-mint.

Lock-and-mint

Lock-and-mint occurs when the user sends funds from the original chain to a destination chain. Tokens sent to RenVM are “locked” in custody. Once the assets are confirmed to be locked, RenVM will release a minting signature to the user, which allows the user to mint a 1:1 tokenized version of the asset on the destination chain. The minted assets are redeemable at any time with no minimum quantity.

Burn-and-release

To complement the first transaction, burn-and-release allows users to send their tokens from the destination chain back to the original chain by burning the tokenized version of the asset on the destination chain and receiving the locked assets on a chosen address. As the name suggests, the pegged tokens are “burned” while the RenVM “releases” an equal amount of the underlying asset on the original chain.

Burn-and-mint

Burn-and-mint combines both transactions above. Users can directly move their assets between host chains by burning the pegged assets from one host chain and minting the same amount of pegged assets on another host chain. However, this will require multiple payments and confirmations through the RenVM, making it a slow and costly process.

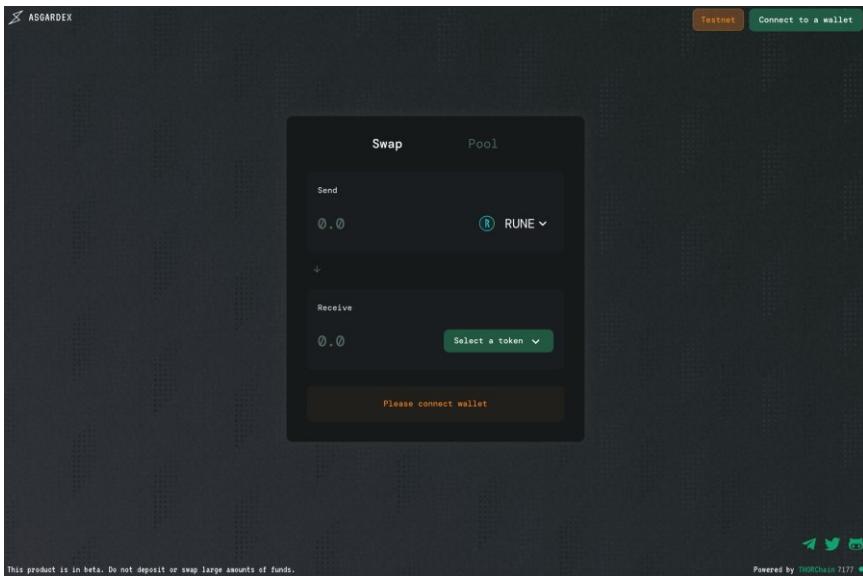
ThorChain



ThorChain is a decentralized liquidity network with an interoperable blockchain that allows cross-chain token swaps in a non-custodial manner. It does not peg or wrap assets but enables users to swap tokens across various Layer 1 blockchains. For instance, traders on ThorChain can seamlessly move their assets from Bitcoin to Ethereum without registering or going through the KYC process of a centralized exchange.

The attraction of ThorChain is that its chain-agnostic feature allows it to swap assets without undergoing some form of conversion. Unlike Ren, there is no 1:1 wrapped Bitcoin (renBTC) created. Instead, we would be able to swap ETH for actual Bitcoin. This is a milestone as previously, the closest representation of Bitcoin in DeFi is in its wrapped form. Thus, ThorChain brings Bitcoin much closer to the heart of the DeFi ecosystem.

Furthermore, as the number of new smart contract platforms grows, such as Solana and Polkadot, the variety of projects and protocols that exist on these blockchains continue to expand at a parabolic rate. The diversity of chains induces the need for a trust-minimized and decentralized way to exchange tokens across different chains.



ThorChain uses the Proof-of-Stake consensus mechanism. It is built on Tendermint where network validators or nodes are required to bond the native token, RUNE. RUNE has a token model that increases in value as the utilization of the network grows. This means that as more liquidity is deposited into ThorChain liquidity pools, RUNE will become more valuable.

RUNE is needed for two fundamental reasons:

- I. In liquidity pools, RUNE acts as a base pair where a 1:1 ratio of ASSET:RUNE is required for staking (e.g., BNB-RUNE or ETH-RUNE). ThorChain does not operate by direct asset transfer; instead, it needs RUNE to move from one asset to another. RUNE is also required to activate ThorChain's Bifrost Protocol, which acts as the bridge that enables multi-chain connectivity. The protocol also tracks the ratio of RUNE to the assets in their Continuous Liquidity Pools (CLP), meaning they also inherit a trustless on-chain price feed for digital assets without relying on third-party oracles.
- II. RUNE is bonded as collateral by node operators to disincentivize malicious actors following a 2:1 bond:stake ratio. RUNE is not intended to be a governance token; ThorChain will be governed

more like Bitcoin, where node operators can determine its future direction. This also means that ThorChain is not limited only to traders but is also used by liquidity providers and node operators.

With a 2:1 bond:stake ratio, combined with the 1:1 pool stake ratio, the amount of RUNE needed would be three times the amount of the non-RUNE assets locked. In other words, this 3:1 ratio represents the intrinsic or minimum value of the RUNE tokens required for the protocol to operate.

Users who utilize ThorChain's cross-chain services between the pools will need to pay fixed network fees and a variable slip fee, to cover gas fees on external services and fast execution. Besides offering a seamless service to traders, users can also become liquidity providers on ThorChain.

Liquidity providers on ThorChain can add liquidity to various pools, which are tied to RUNE in a separate vault. The liquidity pools incentivize any ThorChain participant to supply liquidity in exchange for RUNE rewards, equal to twice the amount of gas used.

As mentioned on ThorChain's website, “liquidity is provided by stakes who earn fees on swaps, turning their unproductive assets into productive assets in a non-custodial manner. Market prices are maintained through the ratio of assets in the pool which traders can arbitrage to restore correct market prices”.

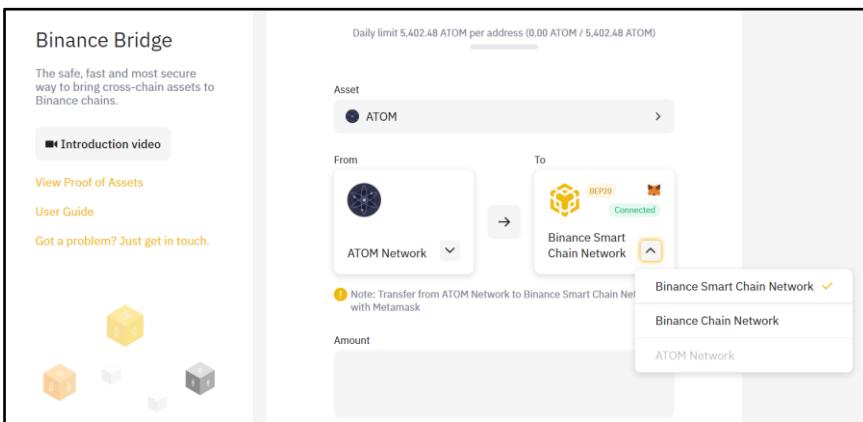
What sets ThorChain apart is its cross-chain feature – it enables users to swap any asset and create a liquidity pool around it, opening a whole new world of possibilities for the DeFi ecosystem.

As of 13 April 2021, THORChain multi-chain Chaosnet is live, along with their decentralized exchange, Asgardex⁷⁸. Users can perform transactions across five active blockchain networks - Bitcoin, Bitcoin Cash, Litecoin, Ethereum, and Binance Chain.

⁷⁸ (2021, April 13). THORChain launch multichain chaosnet | by THORChain ... - Medium. Retrieved May 24, 2021, from <https://medium.com/thorchain/thorchain-launch-multichain-chaosnet-bb9f60008a03>

Binance Bridge

Using the Binance Bridge, users can transfer funds to and from various blockchains such as the Ethereum, Tron, or Binance Smart Chain (BSC) network. Only specific blockchains are supported for each particular asset. For example, native tokens for other blockchains such as Cosmos (ATOM) and Ontology (ONT) are limited to transfers between either Binance Smart Chain, Binance Chain, or their native network. There is a daily limit for how much you can transfer for each asset.



If you are moving assets to the Binance Smart Chain network, you can also opt to swap for some Binance Coin (BNB) as well. Similar to how Ether is used to pay for transaction fees on the Ethereum network, BNB is used to pay for transaction fees on the Binance Smart Chain network. As such, it is advisable to select the option to swap for some BNB, especially if you are a newcomer to the Binance Smart Chain blockchain.

Amount

I want to swap some BNB gas in this order
Please select the amount of swapping BNB

0.5 BNB **1 BNB** 2 BNB

You will receive ≈ 9.83857302 ETH BEP20 + 1 BNB
The final price will depend on the market condition at the time of execution.

Anyswap

Anyswap is a decentralized cross-chain exchange that supports eight different blockchains such as Ethereum, Binance Smart Chain, and Fantom. It offers an all-in-one platform for users to swap or convert their assets onto other blockchains. Users can opt for the conventional method of depositing their assets to mint wrapped tokens or directly perform cross-chain swaps to trade their tokens for another token on a different blockchain.

The screenshot shows the Anyswap DEX dashboard. On the left, there's a sidebar with a purple header labeled "Dashboard" and a list of navigation items: Swap, Pool, Bridge, Farms, Markets, Governance, Network, and Documents. The main area is titled "Dashboard" and features a "My Balance" section. It lists three tokens with their current values:

Coin	Price	Balance	Liquidity	Total Balance	Action
BNB Binance-Peg Binance	\$258.37	-	0	0	Swap
ANY Anyswap-BEP20	\$3.27	-	0	0	Swap
CYC CYCoin	\$0.153287	-	0	0	Swap

A "Search by token name, symbol or address" bar is located above the table. At the bottom of the dashboard, there's a "Show More" button.

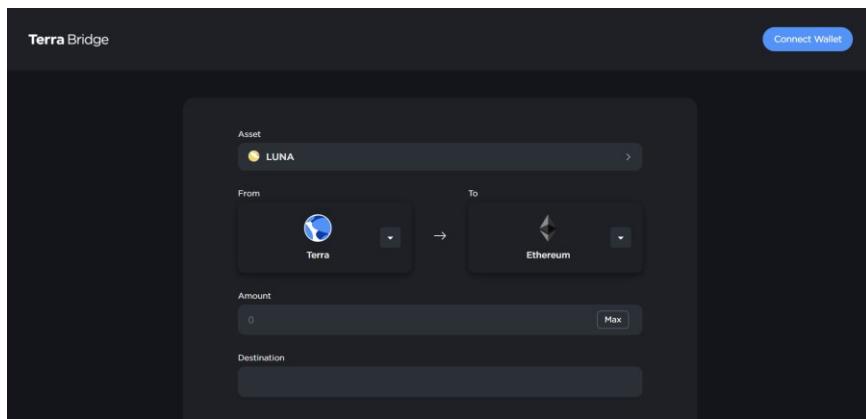
For each blockchain, the Anyswap DEX supports different token pairs, which, as of version 1, are always paired to the network's native token. For example, on the Fantom network, tokens are paired to the FTM token.

The fees charged to use the exchange vary based on the amount of gas used and the type of assets used in the swap. Essentially, users have to pay a 0.4% swap fee in addition to the network's transaction fees. 75% of the fees go to the liquidity providers, and the remainder is sent to Anyswap. Users who perform a transaction between any two non-native assets of their choice are charged the 0.4% fee twice.

Anyswap has risen in popularity ever since Andre Cronje shared his interests and admiration for the cross-chain project. With Total Value Locked of more than \$620 million across their supported blockchains (as of May 2021)⁷⁹, there is still much room for the protocol to grow as more users are willing to embrace and explore other networks. Inspired by Anyswap, Andre Cronje has also released multichain.xyz, a cross-chain protocol that functions similarly with customized pairs for each network.

Terra Bridge

The Terra Bridge is an application for users to send supported Terra assets on the Terra blockchain to and from the Ethereum and Binance Smart Chain networks. Among these assets are LUNA (Terra's native token), Terra stablecoins such as UST (Terra USD), and mirrored assets such as mTSLA (Tesla) and mAAPL (Apple Inc.).



⁷⁹ (n.d.). Network: AnySwap. Retrieved May 24, 2021, from <https://anyswap.net/>

Notable mentions

-  **Multichain.xyz**
Supporting 10 different blockchains, Multichain.xyz allows you to swap a variety of assets such as BNB, ETH and USDC. As of May 24, 2021, it has a Total Value Locked of over \$200 million and supports over 280 tokens⁸⁰.
-  **Matic Bridge**
The Matic Web Wallet Bridge allows users to transfer funds to and from the Polygon network, using Plasma or Proof-of-Stake (PoS) bridge. Depending on the bridge used, only certain assets can be transferred. Withdrawal times may vary.
-  **APYSwap**
APYSwap is a decentralized protocol for exchanging assets across different blockchains with a unique feature - it functions for non-EVM compatible blockchains as well, allowing users to transfer funds from Ethereum, Binance Smart Chain and Huobi ECO Chain to Solana and vice versa.

Associated Risks

Although bridges and cross-chain protocols are becoming a vital step in improving the connectivity between different blockchain networks, users should also be aware of the existing problems that may occur when interacting with these protocols.

Besides the inherent risks that come with the nature of smart contracts and their exposure to faulty code and exploits, users have to ensure that their original assets are truly locked on the native chain before minting pegged assets on other chains. If the original tokens could be freely unlocked and

⁸⁰ (n.d.). Multichain.xyz Stats. Retrieved May 24, 2021, from <https://multichain.xyz/stats>

used by others, the minted assets will become worthless since they cannot be used for redemption back on the original chain.

Users also need to be aware of the varying smart contracts of the tokens for different networks. Although most tokens have the same contract address across multiple blockchains, it is still important to check that you are indeed getting the same version of the token or a pegged equivalent when attempting to perform transfers or cross-chain deposits.

Conclusion

As alternative chains are beginning to see more and more traction, bridges and multi-chain protocols will be more important than ever before. Whether through proxy tokens or seamless swapping between native tokens, it is clear that more users will continue to experiment and cross over into blockchains that may offer different services or lower fees. As such, improving the safety and user experience of these protocols is critical to ensure that DeFi is truly for everyone, on every network.

Recommended Readings

1. Ren Protocol Review
<https://defirate.com/ren-protocol/>
2. An In-depth Guide to Thorchain's Liquidity Pools
<https://medium.com/thorchain/an-in-depth-guide-to-thorchains-liquidity-pools-c4ea7829e1bf>
3. Documentation on Binance Bridge v2
<https://docs.binance.org/smart-chain/guides/bridge-v2.html>
4. Anyswap DEX User Guide
<https://anyswap-faq.readthedocs.io/en/latest/>
5. User Guide for Interchain Transfers on Terra's Shuttle Bridge
<https://docs.anchorprotocol.com/user-guide/interchain-transfers>
6. Guide to use Matic Bridge by Matic Network
<https://blog.matic.network/deposits-and-withdrawals-on-pos-bridge/>
7. How to Add Networks Using Chainlist.org
<https://metamask.zendesk.com/hc/en-us/articles/360058992772-Add-Network-Custom-RPC-using-Chainlist-in-the-browser-extension>

CHAPTER 15: DEFI EXPLOITS

Exploring DeFi is risky; smart contract hacks happen all the time. In 2020 alone, there were at least 12 high-profile DeFi hacks, draining away no less than \$121 million in funds from DeFi protocols.



Source: CoinGecko 2020 Yearly Report

No one - not even the best smart contract auditors - can fully predict what will happen with deployed smart contracts. With billions of dollars of funds sitting on smart contracts, you can be sure that the most brilliant hackers are constantly looking to exploit and profit from security weaknesses.

The big risk for DeFi is that as projects leverage the composable nature of DeFi and build on top of one another, the complexity of DeFi applications increases exponentially, making it harder for smart contract auditors to spot weaknesses. DeFi application developers have to ensure that cybersecurity auditors constantly check their codes to reduce any possibility of exploits because the consequences of mistakes will be huge financial losses.

In this chapter, we will look at the causes of hacks, flash loans, potential solutions to reduce losses from hacks, and some tips for individuals to avoid losing funds in DeFi exploits.

Causes of Exploits

Below we will look at some common causes of exploits. The list is not meant to be exhaustive.

1. Economic Exploits/Flash Loans

Flash loans allow users to leverage nearly limitless capital to carry out a financial transaction as long as the borrower repays the loan within the same transaction. It is a powerful tool that allows one to maneuver economic attacks that used to be constrained by capital requirements. With flash loans, having the right strategy is the only requirement to exploit opportunities.

Almost all DeFi hacks utilized flash loans. We will look into it with more details in the next section.

2. Code in Production Culture

Spearheaded by Andre Cronje, the founder of Yearn Finance, many DeFi projects follow the ethos of test-in-production instead of maximizing security and testing to speed up the pace of product development. Having audits on every release will significantly extend the time required to bring any product updates to market.

One of the main competitive advantages of DeFi is that developers can iterate much faster, pushing the boundaries of financial

innovations. However, not every project can afford to have audits, especially when the project has yet to achieve any traction. Despite having multiple audits, hackers still manage to exploit some projects, suggesting that having audits may not be sufficient to prevent all hacks.

3. Sloppy Coding and Insufficient Audits

In a bull market, many project teams feel pressured to move fast and take shortcuts to release their products quicker. Some may decide to skip audits altogether to have the first-mover advantage and only conduct audits several months after the products are live.

There are also plenty of “forks” - new projects that use the same code as other established projects. Launched without a complete understanding of how the code works, they are treated as a quick cash grab, resulting in many exploits.⁸¹

4. Rug Pull (Inside Jobs)

In the DeFi space, it is not uncommon for projects to launch with anonymous teams. Some do so to avoid the scrutiny of regulators due to an uncertain regulatory climate. However, others chose to be anonymous as they have bad intentions. There have been many instances where anonymous teams conducted an inside job and intentionally left a bug, which is exploited to steal from unsuspecting users.

The crypto community does not alienate projects launched by anonymous founders, seeing how the first cryptocurrency, Bitcoin, was also founded by an unknown person. Users evaluate projects based on the code produced, not who or where the developers are from. This is aligned with the decentralization ethos of open software.

⁸¹ “Saddle Finance - REKT - Rekt News.” 20 Jan. 2021, <https://www.rekt.news/saddle-finance-rekt/>. Accessed 4 May. 2021.

Ideals aside, if an exploit were to occur on a protocol launched by an anonymous team, the chances of no recourse are high as it is hard to find the real-world identity of the developers.

5. Oracle Attacks

DeFi protocols need to know asset prices to function correctly. For example, a lending protocol needs to know the asset price to decide whether to liquidate the borrowers' position.

Therefore, as an indispensable part of DeFi infrastructure, oracles may be subject to heavy manipulation. For example, we mentioned in [Chapter 12](#) how the exploit of MakerDAO's vault caused unnecessary vaults' liquidation that totaled more than \$8 million worth of ETH in losses.

6. Metamask Attack

As the main interface to every Ethereum application, it is no surprise that Metamask has become a primary attack target. The ConsenSys team has been thorough in security, and to date, there have been no widespread exploits.

However, there were a few high profile attacks:

- \$59 million loss through the EasyFi project's admin MetaMask account⁸²
- \$8 million loss through the personal wallet of Nexus Mutual's founder⁸³

⁸² "EasyFi - REKT - Rekt News." 20 Apr. 2021, <https://www.rekt.news/easyfi-rekt/>. Accessed 4 May. 2021.

⁸³ "Rekt - Nexus Mutual - Hugh Speaks Out - Rekt News." 23 Dec. 2020, <https://www.rekt.news/nxm-hugh-speaks-out/>. Accessed 4 May. 2021.

Flash Loans

What are Flash Loans?

Flash loans are loans where users can borrow funds without any collateral so long as the user pays back the loan in the same transaction. If the user does not repay the flash loan in the same transaction, the transaction will automatically fail, incurring a loss in the transaction fee and ensuring that the flash loan will not take place.⁸⁴ Flash loans are offered by various DeFi protocols such as Aave and dYdX.

Flash loans have three main characteristics that make them stand out as compared to a normal loan:⁸⁵

- **No default risk:** Flash loans are repaid within the same transaction. Therefore there is no risk of getting defaulted.
- **No collateral:** Borrowers can take the loan without posting any collateral or credit check as long as they can repay the loan within one transaction.
- **Unlimited loan size:** Users can borrow any amount up to the total liquidity available from the DeFi protocols.

As of April 2021, the execution of flash loans is not user-friendly, as you must execute it by writing a smart contract code. Thus, it is more accessible to software programmers rather than the average layman.

Indeed, some programmers often exploit this factor and launch what we call a “flash loan attack”, especially when no collateral is needed. One of the most famous flash loan attacks occurred on Harvest Finance, resulting in a \$24 million loss.⁸⁶

⁸⁴ “Open Source DeFi Protocol | FlashLoan - Aave.” <https://aave.com/flash-loans/>. Accessed 4 May. 2021.

⁸⁵ Qin, K., Zhou, L., Livshits, B., & Gervais, A. (2021, March 20). Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. <https://arxiv.org/pdf/2003.03810.pdf>

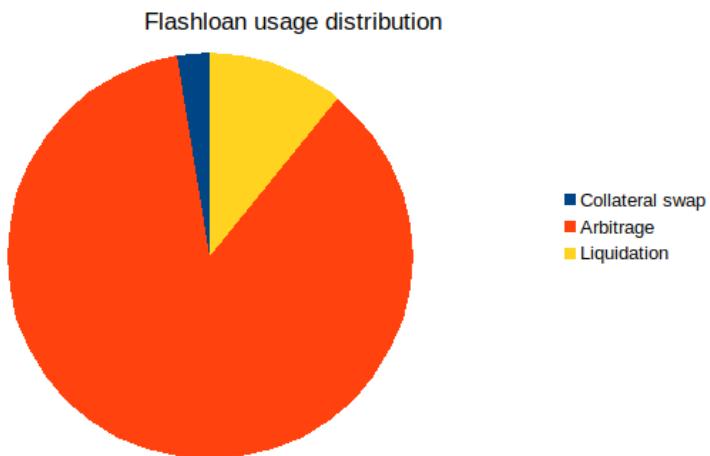
⁸⁶ “Harvest Finance: \$24M Attack Triggers \$570M ‘Bank Run’ in Latest” 26 Oct. 2020, <https://www.coindesk.com/harvest-finance-24m-attack-triggers-570m-bank-run-in-latest-defi-exploit>. Accessed 4 May. 2021.

The table below shows the fees incurred on the major protocols that offer flash loans:

Protocols	Flash Loan Fee
Aave	0.09%
dYdX	1 Wei (10^{-18}) ETH
bZx	None
Uniswap v2	0.3%

Usage of Flash Loans

The following chart shows the composition of all the Flash Loans usage from Aave⁸⁷:



We can see that flash loans are mainly used for arbitrage purposes. Arbitrage is the act of exploiting price differences between markets to make a profit. For instance, let's say we discover a considerable price difference for WBTC on two different decentralized exchanges. We can use a flash loan to borrow a substantial amount of WBTC without any collateral to profit from the price difference.

⁸⁷ "Flash Loans, one month in. Balancing fees and first usage ... - Medium." 12 Feb. 2020, <https://medium.com/aave/flash-loans-one-month-in-73bdc954a239>. Accessed 12 May. 2021.

The second usage of flash loans is for loan liquidation. There is usually a penalty for the borrowers if they let the protocol liquidate their position. When the market has substantial price actions, borrowers can choose to obtain flash loans and self-liquidate their positions, avoiding the penalty fees.

Let's look at an example where we borrow DAI from Maker with ETH as collateral. When the price of ETH falls significantly, it may get near the liquidation level for our DAI loan. We may not have ETH to increase our collaterals nor DAI to repay the loan. What we can do is take a DAI flash loan to repay the Maker loan. We can then swap a portion of the withdrawn ETH collateral to DAI to repay the flash loan instantly. Using this method, we will keep the remaining ETH without paying the liquidation penalty.

Lastly, flash loans can also be used to execute a collateral swap. For instance, if we have a DAI loan in Compound with ETH as collateral, we can swap the ETH collateral to WBTC collateral using a flash loan. This allows us to change our risk profiles easily without having to go through multiple transactions.

Executing flash loans still requires considerable technical knowledge and has high entry barriers to those who do not know how to code. However, there is a third-party app that makes the execution of flash loans accessible for average users – this platform is called Furucombo.

Flash Loan Protocol: Furucombo

Furucombo is a platform that allows anyone to create arbitrage strategies using flash loans. Thanks to its drag-and-drop tool that enables end-users to build and customize different DeFi combinations, the barriers to entry for assembling money-legos are lowered. Do note that Furucombo does not find arbitrage opportunities for you.

To use Furucombo, users need to set up input/outputs and the order of the transactions, and it will bundle all the cubes into one transaction for execution. An example of how an arbitrage transaction can be carried out is shown below:



1. Obtain a 15,000 DAI flash loan from Aave.
2. Swap DAI to yCRV using 1inch.
3. Exchange yCRV back to DAI using Curve. Due to price differentials, you end up with 15,431 DAI, more DAI than before.
4. The loan amount of 15,013 DAI including Aave's flash loan fee is repaid to Aave. The user is left with a profit of 418 DAI. All of these steps are executed within one transaction.

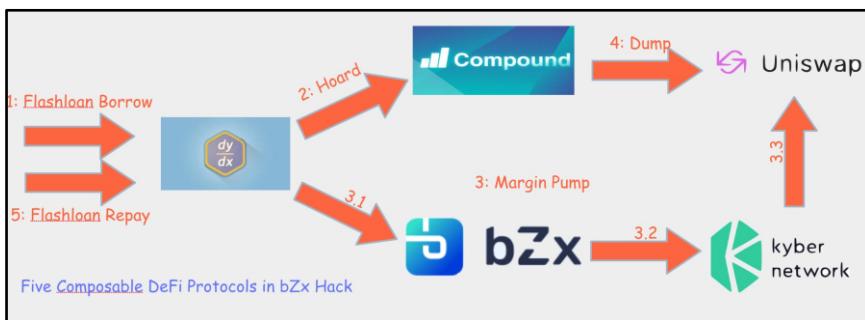
Furucombo does not require any upfront funds nor charge fees for users to build “combos” and make arbitrage trades using flash loans on the platform. All you need is ETH in your wallet to pay for the gas fee.

Users are advised to trade at their own risk as arbitrage opportunities are not always available on Furucombo, and a combo may fail if the price difference no longer exists. Users face the risk of paying for the transaction fees regardless of the outcome.

Case Study: bZx Flash Loans Hack

On 15 February 2020, a transaction took place on the Ethereum blockchain that was considered unique at the time. A profit of approximately \$360,000 was achieved within one block and in one transaction in just under a minute.

This transaction caught the crypto community's attention and was widely analyzed.⁸⁸ The gain was achieved via an initial nearly-risk-free loan in the form of a flash loan, subsequently followed by a series of arbitrage between different decentralized exchanges.



Source: Peckshield⁸⁹

1. Flashloan Borrow

First, a flash loan of 10,000 ETH was taken from dYdX.

2. Hoard

Half of those ETH (5,500 ETH) were staked as collateral in Compound to borrow 112 WBTC.

3. Margin Pump

1,300 ETH is deposited into bZx to short ETH in favor of WBTC using 5x leverage. 5,637 ETH loaned from bZx was used to swap for 51 BTC using KyberSwap. According to the KyberSwap algorithm, the best price was offered by Uniswap. However, due to the low liquidity, the swap drove up the exchange rate of 1 WBTC to around 109.8 WETH, roughly triple the normal conversion rate during the period.

4. Dump

The attacker sold the borrowed 112 WBTC in Uniswap after the price increase, yielding 6,871 ETH with a conversion rate of 1 WBTC = 61.2 WETH.

⁸⁸ “Exploit During ETHDenver Reveals Experimental Nature ... - CoinDesk.” 15 Feb. 2020, <https://www.coindesk.com/exploit-during-ethdenver-reveals-experimental-nature-of-decentralized-finance>. Accessed 4 May. 2021.

⁸⁹ “bZx Hack Full Disclosure (With Detailed Profit Analysis ... - PeckShield.” <https://peckshield.medium.com/bzx-hack-full-disclosure-with-detailed-profit-analysis-e6b1fa9b18fc>. Accessed 4 May. 2021.

5. Flash Loan Profit

With an unused 3,200 ETH and 6,871 ETH from the sale, the attacker paid back the 10,000 ETH flash loan with a profit of 71 ETH.

Protocol	Amount	Asset	Type
dYdX	-10,000	ETH/WETH	Debt
Compound	+5,500	ETH/WETH	Collateral
Compound	-112	WBTC	Debt
bZx	+1,300	ETH/WETH	Collateral
bZx	-5,637	ETH/WETH	Debt
bZx	+51	WBTC	Collateral
Accounts	Amount	Asset	Type
-	+3,200	ETH/WETH	Balance
-	+6,871	ETH/WETH	Balance

6. Total Profit

The Compound position was still in profit. As the average market price of 1 WBTC was 38.5 WETH, the attacker can get 112 WBTC with roughly 4,300 ETH. In total, the attacker gained 71 WETH + 5,500 WETH - 4,300 ETH = 1,271 ETH, roughly \$355,880 (assuming the ETH price of \$280).

The event above not only demonstrates the possibility of extreme capital gains by manipulating the price of other assets but that there were also no other costs for the borrower besides a relatively low protocol fee. The only condition faced by the borrower was that the loan was to be repaid within the same transaction. Thus, the very concept of uncollateralized loans opens up a wide range of opportunities in the space.

Flash Loan Summary

Flash loans can be a double-edged sword. On one hand, its novel use of smart contracts brings convenience and advancements to the DeFi ecosystem - traders without much capital can launch arbitrage and liquidation strategies with flash loans without the need for a large capital base.

On the other hand, hackers can use flash loans to launch flash loan attacks, vastly enhancing their profits since no collateral is required. Just like any tool, flash loans can be used for both good and bad purposes.

In our view, flash loan attacks utilized for ill-purposes have strengthened the whole DeFi ecosystem as projects improve their infrastructure to prevent future attacks from occurring. As flash loans are still at a nascent stage, the attacks can be seen as a silver lining that mitigates the kinks in the DeFi ecosystem and makes it more antifragile.

Solutions

Having only smart contract audits is not enough to prevent exploits. Projects need to do more, and they are now looking for alternatives to ensure the safety of funds deposited in their protocols. Below are some of the possible solutions:

Internal Insurance Fund

Several projects have decided to use their native token as the risk backstop. Examples include:

- Maker minted MKR back on Black Thursday to cover for liquidation shortfall in DAI.
- Aave rolled out stAAVE to cover any potential shortfall for the depositors.
- YFI collateralized their token and borrowed DAI to pay back hacked funds.⁹⁰

Insurance

Being one of the new kids on the block, Unslashed Finance offered protocol-level covers to LIDO and Paraswap for their users. This opens up the possibility for protocols to buy covers for their users.

⁹⁰ "Yearn.Finance puts expanded treasury to use by repaying victims of" 9 Feb. 2021, <https://cointelegraph.com/news/yearn-finance-puts-expanded-treasury-to-use-by-repaying-victims-of-11m-hack>. Accessed 12 May. 2021.

Bug Bounty

Projects are increasingly leveraging Immunefi to list bug bounties,⁹¹ encouraging hackers to claim rewards for finding bugs rather than exploiting them. The highest bounties offer up to \$1.5 million. However, with higher potential rewards from hacking, whether this will deter the hackers remains to be seen.

Other Possible Solutions

- Industry-wide insurance pool

There can be an industry-wide insurance pool where every DeFi protocol chips in part of their earnings or pays a fixed fee. The pool is expected to pay out claims when one of the members experiences a hack. This is similar to the idea of the Federal Deposit Insurance Corporation (FDIC).

- Auditors to have skin in the game

This idea proposed auditors to stake on DeFi insurance platforms such as Nexus Mutual. As stakers, if the protocols are hacked, then the auditors will experience loss. This implementation is expected to align the interest of auditors and the project.

Tips for Individuals

Besides smart contract hacks, you may also be exposed to various hacking attempts. Below are some steps you can take to minimize risks and reduce the chances of getting hacked.

Don't Give Smart Contracts Unlimited Approval

Interacting with DeFi protocols usually requires you to give smart contracts access and consent to spend your wallet's funds. Usually, for convenience purposes, DeFi protocols request that the default approval be set as infinity, meaning the protocol has unlimited access to the approved asset on your wallet.

⁹¹ (n.d.). Immunefi. Retrieved May 23, 2021, from <https://immunefi.com/>

Giving unlimited approval to spend approved assets on your wallet is usually a bad idea because a malignant smart contract may exploit this to drain funds from your wallet.

On 27 February 2021, a hacker used a fake smart contract and tricked Furucombo into thinking that Aave v2 had a new implementation. The attack exploited large wallets with unlimited token approvals and drained these wallets by transferring funds to a hacker-controlled address.

This attack saw Furucombo users experiencing losses totaling up to \$15 million.⁹² Even the lending protocol, Cream Finance, made the mistake of having unlimited approvals and lost \$1.1 million from their treasury in this attack.⁹³

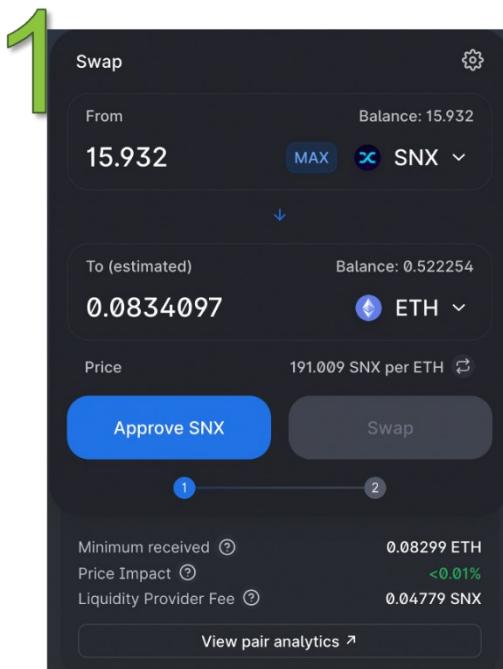
Thus, it is a good idea to manually change the approved amount for each transaction to prevent giving DeFi protocols unlimited spending permission during token approvals. Even though this will result in an additional transaction for each subsequent DeFi interaction and incur higher transaction fees, you can reduce the risk of wallets being drained in a smart contract exploit attack.

To manually change the approved amount, follow the step-by-step guide below.

⁹² “Furucombo Post-Mortem March 2021. Dear Furucombo ... - Medium.” 1 Mar. 2021, <https://medium.com/furucombo/furucombo-post-mortem-march-2021-ad19afd415e>. Accessed 12 May. 2021.

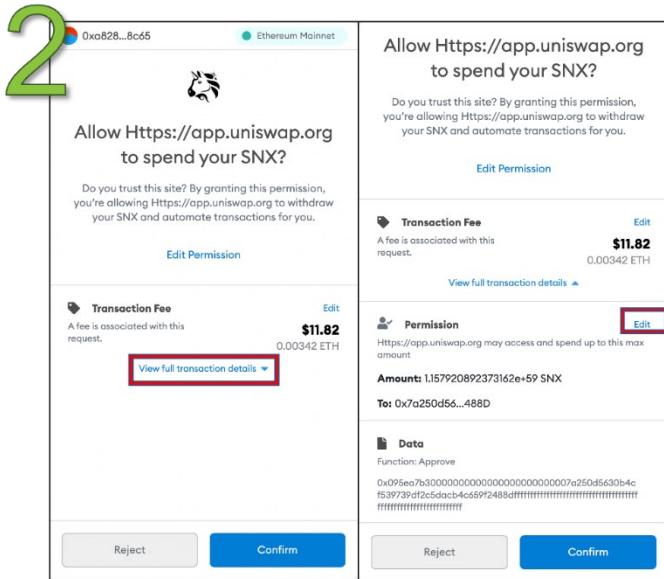
⁹³ “defiprime on Twitter: “⚠️ Just in: Furucombo exploited ⚠️ If you” 27 Feb. 2021, <https://twitter.com/defiprime/status/1365743488105467905>. Accessed 12 May. 2021.

Don't Give Smart Contracts Unlimited Approval: Step-by-Step Guide



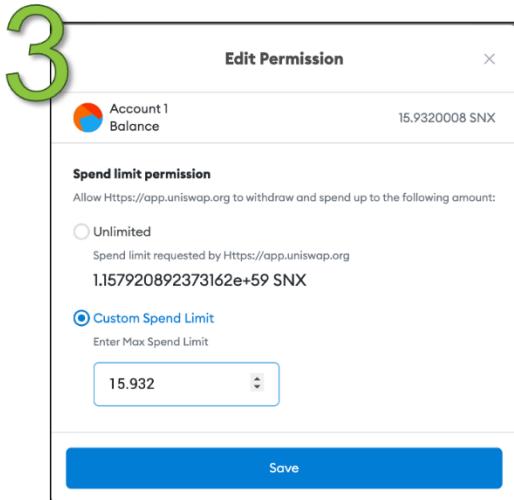
Step 1

- One of the most common instances that require approvals is swapping.
- In the example, we plan to swap 15.932 SNX to 0.0834 ETH on Uniswap.
- Click “Approve SNX”



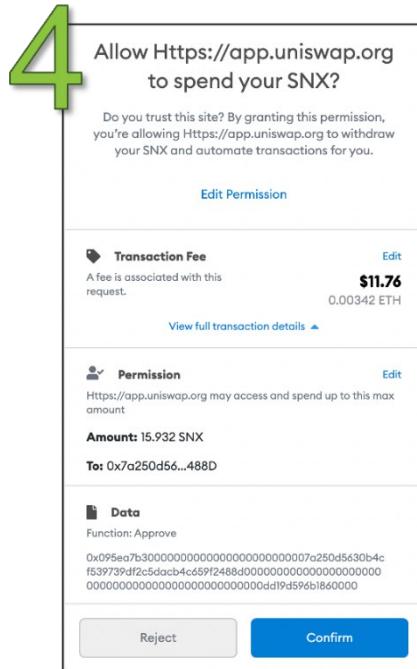
Step 2

- The left image is the default window that we will see.
- Click “View full transaction details”
- Then we will be presented with the right image. Click “Edit”.



Step 3

- Choose “Custom Spend Limit”.
- A lot of applications choose the “Unlimited” option in default.
- Key in the amount that we want to spend, in this example, that’s 15.932 SNX.



Step 4

- Under the “Permission” section we can see that the figures are updated.
- Click “Confirm” and pay the transaction fee.

Revoking Unlimited Approvals from Smart Contracts

If you have previously given DeFi protocols unlimited spending approvals, there are two options for you. The easier option is to move all your funds to a new Ethereum address, and you will get a fresh restart without taking on any of the risks associated with your previous Ethereum address.

However, if moving funds out of your existing Ethereum address is not possible, you can check the list of all previous approvals using the Token Approval tool provided by Etherscan.⁹⁴

⁹⁴ (n.d.). Token Approvals @ etherscan.io. Retrieved May 23, 2021, from <https://etherscan.io/tokenapprovalchecker>

Contract	Approved Spender	Approved Amount	Last Updated (UTC)	Revoke
Compound Dai	[REDACTED]	Unlimited cDAI	2021-01-27 02:37:43	

On this page, you can see all the approvals that you have previously granted to smart contracts. You should revoke all approvals with Unlimited approved amounts, especially protocols that you no longer interact with. Do note that revoking each approval requires a smart contract interaction itself and will incur transaction fees.

Use a Hardware Wallet

A hardware wallet is a physical device used solely for storing cryptocurrencies. Hardware wallets keep private keys separate from internet-connected devices, reducing the chances of your wallet being compromised.

In hardware wallets, the private keys are maintained in a secure offline environment, even if the hardware wallet is plugged into a computer infected with malware. While hardware wallets can be physically stolen, it is not accessible if the thief does not know your passcode. In the unfortunate event that your hardware wallet is damaged or stolen, you will still be able to recover your funds if you had created a secret backup code prior to the loss.

The top hardware wallets manufacturers are Ledger and Trezor, though more have entered the industry.

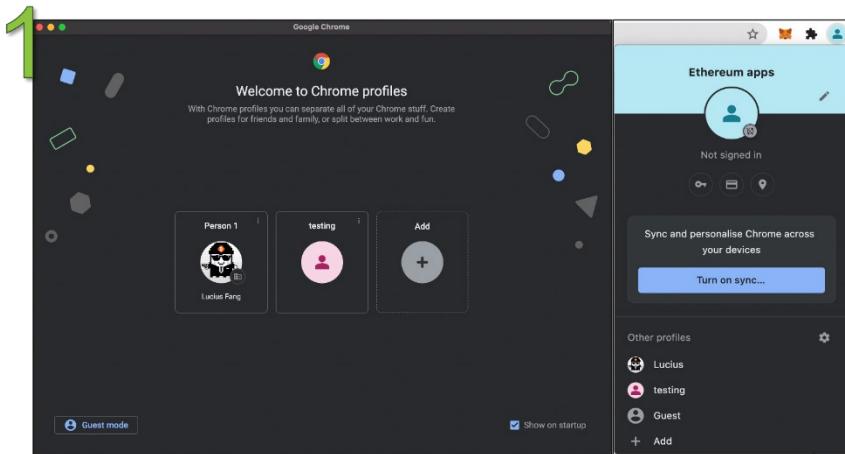
Use a Separate Browser Profile

Although browser extensions are helpful and make you more effective and productive in your work, you should always be worried about malicious browser extensions causing trouble with your cryptocurrency experience.

If you accidentally installed a malicious browser extension, it may snoop on your Metamask keys and become an attack vector on your funds. One method to improve your security is to create a separate browser profile on your Google Chrome or Brave browser. In that new browser profile, install only the Metamask extension. By doing so, you reduce the risk of a malicious browser extension siphoning out funds from your wallet.

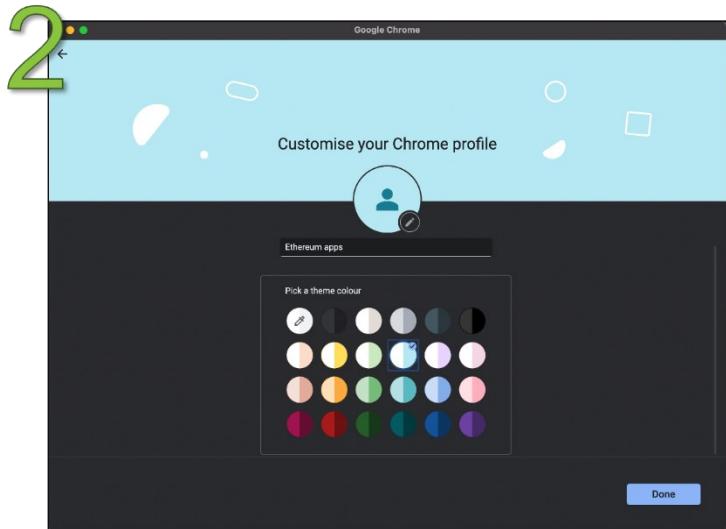
Here is a step-by-step guide to creating a new browser profile on Google Chrome.

Separate Browser Profile: Step-by-Step Guide



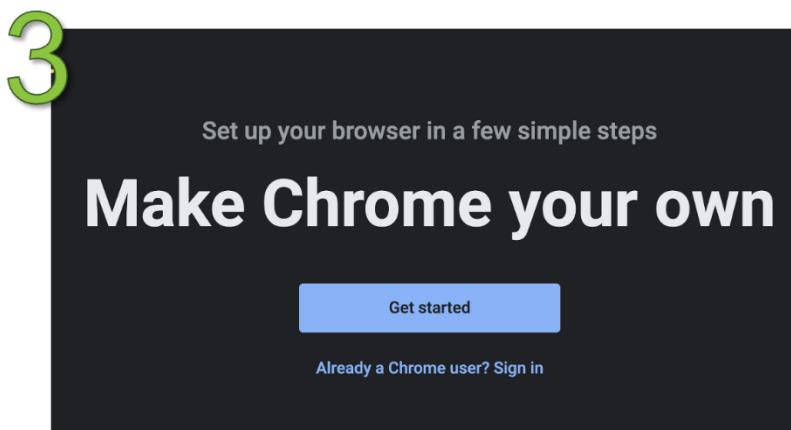
Step 1

- Opening the Chrome browser will lead to the above page. Click “Add”.
- Alternatively you can go to the upper right-hand corner and click on the profile icon. Click “+ Add” at the very bottom.



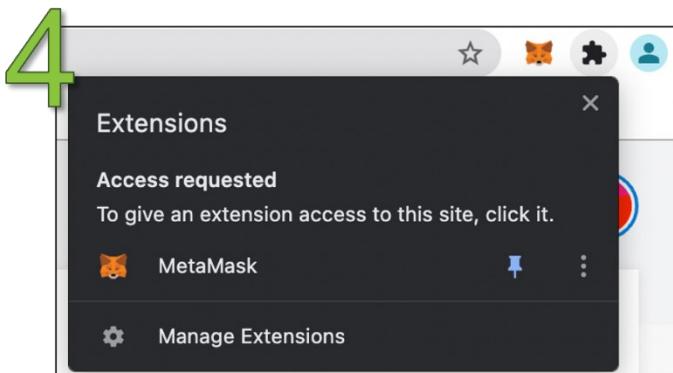
Step 2

- Pick the name and color. Then click “Done”



Step 3

- Sign in a different Chrome account if you have one. If not, just click “Get started”.



Step 4

- Download the Metamask extension and make it the only extension in this profile.

Conclusion

The DeFi space is still very much an experimental ground for various financial innovations. As such, things may and can go wrong. Do be aware of the risks of using new DeFi applications, especially those that are not battle-tested.

Always do your research before using any DeFi protocols. In most cases, once a mistake takes place, there is no recourse to the losses incurred. Even in the event of remuneration by the protocol, the losses usually exceed the amount of compensation received.

That said, because of the risks involved, the returns from participating in DeFi activities are high. To not miss out on the high returns offered by DeFi, you can opt to hedge some risks by buying insurance or put options.

The DeFi space is still maturing. As we go along, we expect more DeFi protocols to launch with better safeguards and insurance funds incorporated into their operating models.

Recommended Readings

1. To be your own bank with Metamask
<https://consensys.net/blog/metamask/metamask-secret-seed-phrase-and-password-management/>
2. How (Not) To Get Rekt – DeFi Hacks Explained
<https://finematics.com/defi-hacks-explained/>
3. DeFi Security: With So Many Hacks, Will It Ever Be Safe?
<https://unchainedpodcast.com/defi-security-with-so-many-hacks-will-it-ever-be-safe/>
4. News on hacks and exploits
<https://www.rekt.news/>

CHAPTER 16: THE FUTURE OF FINANCE

DeFi's mission is clear: reinventing traditional finance's infrastructure and interface with greater transparency, accessibility, efficiency, convenience, and interoperability.

As of April 2021, the Total Value Locked (TVL) in DeFi applications has reached \$86 billion - 86 times larger since we last published our *How to DeFi: Beginner* (First Edition) in March 2020. Back then, the TVL within DeFi just hit \$1 billion.

Here is a quick summary of DeFi milestones:

- 2018: TVL increased 5 times from \$50 million to \$275 million
- 2019: TVL increased 2.4 times to \$667 million
- 2020: TVL increased 23.5 times to \$15.7 billion
- 2021: TVL increased 5.5 times to \$86.05 billion (as of April 2021)

Through both of our How to DeFi books, you can see that DeFi, in itself, is reimagining the way global financial systems operate. As we covered in all our chapters, various financial primitives are already live, such as decentralized exchanges, lending, insurance, and derivatives. Regardless of location and status, DeFi has made it possible for anyone in the world to access financial services as long as they have Internet access.

Although most of the DeFi protocols we covered are Ethereum natives, we already see the exponential growth of users and projects on other blockchain networks like Binance Smart Chain, Solana, Polygon, and Fantom.

Reinventing legacy finance is about more than just the tech. It also means reinventing the culture. At its heart, DeFi represents a transparent and open-source movement with an extremely powerful culture that continues its mission to create tens of trillions of dollars in value. This culture is what helps to shape and legitimize DeFi.

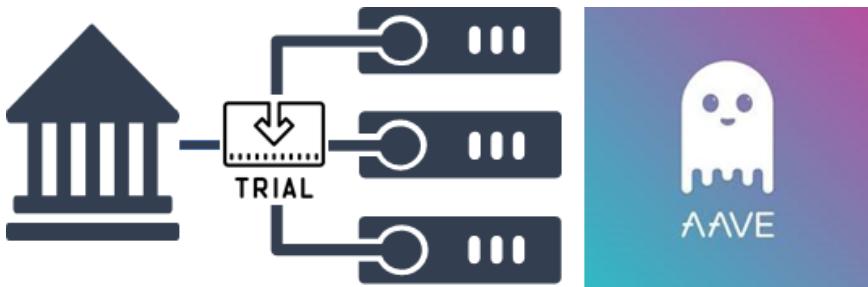
Great minds have come together and attempted to solve some of the most pressing issues that plague traditional finance. The result combines traditional finance principles, innovation, and blockchain technology while offering superior financial products and services.

How Long Before Institutions Build on These Networks?

We have already seen instances of institutions building on DeFi. For example, Visa announced in the first quarter of 2021 that it will soon start settling transactions with USDC on Ethereum.⁹⁵ In May 2021, Aave built a private pool for institutions as a practice ground before entering the DeFi ecosystem.⁹⁶ Both are excellent examples of collaborative learning between the two financial systems and acts as a precursor to institutions' direct involvement with DeFi.

⁹⁵ (2021, March 29). Visa Will Start Settling Transactions With Crypto Partners In USDC Retrieved May 24, 2021, from <https://www.forbes.com/sites/ninabambysheva/2021/03/29/visa-to-start-settling-transactions-with-bitcoin-partners-in-usdc/>

⁹⁶ (2021, May 12). DeFi lending platform Aave reveals ‘permissioned pool’ for institutions. Retrieved May 24, 2021, from <https://cointelegraph.com/news/defi-lending-platform-aave-reveals-private-pool-for-institutions>



```
// -----
// PROTOCOL GLOBAL PARAMS
// -----  
  
export const CommonsConfig: ICommonConfiguration = [  
  MarketId: 'Commons',  
  ATokenNamePrefix: 'Aave Pro market',  
  StableDebtTokenNamePrefix: 'Aave Pro stable debt ',  
  VariableDebtTokenNamePrefix: 'Aave Pro variable debt ',  
  SymbolPrefix: '',  
  ProviderId: 0, // Overridden in index.ts  
  ProtocolGlobalParams: {  
    TokenDistributorPercentageBase: '10000',  
    MockUsdPriceInWei: '5848466240000000',  
    IlAddress: '0x10F7Fc1F91Ba351f0R629c5947AD60h0A3c05h96'
```

A screenshot of lines of code on Aave Pro's smart contract.

Source: <https://twitter.com/StaniKulechov/status/1394390461968633859/photo/1>

Where Does This Take Us in the Next 5 to 10 Years?

It is difficult to say how things will be in the future, but we would like to think of DeFi as a technological movement that challenges the status quo of traditional finance. Similar to how the Internet has made many inventions obsolete by revolutionizing the way we communicate and share information, DeFi will do the same for finance and take advantage of a global network to create a more transparent and efficient financial system.

Development takes time, but the pace of innovation in DeFi has been moving at breakneck speed. As of April 2021, the Total Value Locked in DeFi has grown over 1,700 times larger than what it was four years ago in 2018.

One of the big reasons we got here is thanks to the DeFi developers and community who have been relentlessly building, building, and building despite the competitive environment in the space. For that, we want to say thank you all for making open finance possible.

CLOSING REMARKS

Congratulations on making it this far! From the current state of DeFi to decentralized exchanges and exploits, our journey through DeFi in this book has come to a close. However, this is not the end, as there will always be new things to learn and new protocols to explore.

By now, you should have a deeper understanding of DeFi and how it works. You should know that DeFi moves very fast and is infinitely complex. By the time we publish this book, some of the information might already be outdated!

Nevertheless, we hope that *How to DeFi: Advanced* book will be a core reference point in your DeFi journey. May it guide you on your never-ending quest to explore the many rabbit holes of DeFi.

APPENDIX

CoinGecko's Recommended DeFi Resources

Analytics

- DefiLlama - <https://defillama.com/home>
- DeBank - <https://debank.com/>
- DeFi Prime - <https://defiprime.com/>
- DeFi Pulse - <https://defipulse.com/>
- Dune Analytics - <https://duneanalytics.com/home>
- LoanScan - <http://loanscan.io/>
- Nansen - <https://www.nansen.ai/>
- Token Terminal - <https://www.tokenterminal.com/>
- The Block Dashboard - <https://www.theblockcrypto.com/data>

News Sites

- CoinDesk - <https://www.coindesk.com/>
- CoinTelegraph - <https://cointelegraph.com/>
- Decrypt - <https://decrypt.co/>
- The Block - <https://www.theblockcrypto.com/>
- Crypto Briefing - <https://cryptobriefing.com/>

Newsletters

- CoinGecko - <https://landing.coingecko.com/newsletter/>
Bankless - <https://bankless.substack.com/>
DeFi Tutorials - <https://defitutorials.substack.com/>
DeFi Weekly - <https://defiweekly.substack.com/>
DeFi Pulse Farmer - <https://yieldfarmer.substack.com/>
Delphi Digital - <https://www.delphidigital.io/research/>
Dose of DeFi - <https://doseofdefi.substack.com/>
Ethhub - <https://ethhub.substack.com/>
Deribit Insight - <https://insights.deribit.com/>
My Two Gwei - <https://mytwogwei.substack.com/>
Messari - <https://messari.io/>
The Defiant - <https://thedefiant.substack.com/>
Week in Ethereum News - <https://www.weekinethereumnews.com/>

Podcast

- CoinGecko - <https://podcast.coingecko.com/>
BlockCrunch - <https://castbox.fm/channel/Blockcrunch%3A-Crypto-Deep-Dives-id1182347>
Chain Reaction - <https://fiftyonepercent.podbean.com/>
Into the Ether - Ethhub - <https://podcast.ethhub.io/>
PoV Crypto - <https://povcryptopod.libsyn.com/>
Uncommon Core - <http://uncommoncore.co/podcast/>
Unchained Podcast - <https://unchainedpodcast.com/>
Wyre Podcast - <https://blog.sendwyre.com/wyretalks/home>

Youtube

- Bankless -
<https://www.youtube.com/channel/UCAl9Ld79qaZxp9JzEOwd3aA>
Chris Blec - <https://www.youtube.com/c/chrisblec>
DeFi Dad -
<https://www.youtube.com/channel/UCAtItl6C7wJp9txFMbXbSTg>
Economics Design - <https://www.youtube.com/c/EconomicsDesign>
The Defiant - <https://www.youtube.com/channel/UCI0J4MLEdLP0-UyLu0hCktg>

Yield TV by Zapper -

<https://www.youtube.com/channel/UCYq3ZxBx7P2ckJyWVDC597g>

Bankless Level-Up Guide

<https://bankless.substack.com/p/bankless-level-up-guide>

Projects We Like Too

Dashboard Interfaces

Zapper - <https://zapper.fi/dashboard>

Frontier - <https://frontierwallet.com/>

InstaDapp - <https://instadapp.io/>

Zerion - <https://zerion.io/>

Debank - <https://debank.com/>

Decentralized Exchanges

Uniswap - <https://uniswap.org/>

SushiSwap - <https://sushi.com/>

Balancer - <https://balancer.exchange/>

Bancor - <https://www.bancor.network/>

Curve Finance - <https://www.curve.fi/>

Kyber Network - <https://kyberswap.com/swap>

Dodo - <https://dodoex.io/>

Exchange Aggregators

1inch - <https://1inch.exchange/>

Paraswap - <https://paraswap.io/>

Matcha - <https://matcha.xyz/>

Lending and Borrowing

Maker - <https://oasis.app/>

Compound - <https://compound.finance/>

Aave - <https://aave.com/>

Cream - <https://cream.finance/>

Oracle and Data Aggregator

Covalent - <https://www.covalenthq.com/>

The Graph - <https://thegraph.com/>

Prediction Markets

Augur - <https://www.augur.net/>

Taxes

TokenTax - <https://tokentax.co/>

Wallet

Metamask - <https://metamask.io/>

Argent - <https://argent.link/coingecko>

Dharma - <https://www.dharma.io/>

GnosisSafe - <https://safe.gnosis.io/>

Monolith - <https://monolith.xyz/>

Yield Optimizers

APY Finance - <https://apy.finance/>

Yearn - <https://yearn.finance/>

Alpha Finance - <https://alphafinance.io/>

References

Chapter 1: DeFi Snapshot

- Bambysheva, N. (2021, March 29). *Visa Will Start Settling Transactions With Crypto Partners In USDC On Ethereum*. Forbes.
<https://www.forbes.com/sites/ninabambysheva/2021/03/29/visa-to-start-settling-transactions-with-bitcoin-partners-in-usdc/>.
- Emmanuel, O. (2021, April 16). Citibank Demystifies MakerDAO and DeFi for Fund Managers. BTCManager.
<https://btcmanager.com/citibank-makerdao-defi-fund-managers/>.
- Franck, T. (2021, March 26). *Fidelity to launch bitcoin ETF as investment giant builds its digital asset business*. CNBC.
<https://www.cnbc.com/2021/03/24/fidelity-to-launch-bitcoin-etf-as-investment-giant-builds-its-digital-asset-business-.html>.
- Kharpal, A. (2021, April 19). *After a bitcoin crackdown, China now calls it an 'investment alternative' in a significant shift in tone*. After a bitcoin crackdown, China now calls it an ‘investment alternative’ in a significant shift in tone. <https://www.cnbc.com/2021/04/19/china-calls-bitcoin-an-investment-alternative-marking-shift-in-tone.html>.
- Manning, L. (2021, April 19). *Coinbase IPO Exceeds All Expectations, Showing More Promise For Bitcoin*. Nasdaq.
<https://www.nasdaq.com/articles/coinbase-ipo-exceeds-all-expectations-showing-more-promise-for-bitcoin-2021-04-19>.
- Schär, F. (2021, February 5). Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets.
<https://research.stlouisfed.org/publications/review/2021/02/05/decentralized-finance-on-blockchain-and-smart-contract-based-financial-markets>.
- Schmitt, L. (2021, April 22). *DeFi 2.0-First Real World Loan is Financed on Maker*. Medium. <https://medium.com/centrifuge/defi-2-0-first-real-world-loan-is-financed-on-maker-fbe24675428f>.

Chapter 2: DeFi Activities

Cryptopedia, S. (2021, April 17). Airdrops: Crypto Airdrops and Blockchain Airdrops. Gemini. <https://www.gemini.com/cryptopedia/airdrop-crypto-giveaway-uniswap>.

Etherscan. (n.d.). Airdrops List. <https://etherscan.io/airdrops>.

Etherscan (n.d) Ethereum Activities Stats. <https://etherscan.io/charts>.

How to add tokens to SushiSwap Exchange as an LP. SushiSwap. (n.d.).
<https://help.sushidocs.com/guides/how-to-add-tokens-to-sushiswap-exchange-as-an-lp>.

Liquidity Bootstrapping FAQ. Balancer. (n.d.).

<https://docs.balancer.finance/smart-contracts/smart-pools/liquidity-bootstrapping-faq>.

Xie, L. (2021, March 13). A beginner's guide to DAOs. Mirror.

https://linda.mirror.xyz/Vh8K4leCGEO06_qSGx-vS5IvgUqhqkCz9ut81WwCP2o.

Zapper: Dashboard for DeFi. (n.d.). <https://zapper.fi/dashboard>.

Chapter 3: Decentralized Exchanges

Balancer Whitepaper. (2019, September 19).

<https://balancer.fi/whitepaper.pdf>.

Bancor. (2021, February 17). Using Bancor Vortex. Medium

<https://blog.bancor.network/using-bancor-vortex-46974a1c14f9>.

Chainlink. (2020, June 29). DeFi Series: More Capital & Less Risk in Automated Market Makers. Chainlink.

<https://blog.chain.link/challenges-in-defi-how-to-bring-more-capital-and-less-risk-to-automated-market-maker-dexs/>.

Appendix

FAQ. Balancer. (n.d.). <https://docs.balancer.finance/getting-started/faq>.

Hasu. (2021, April 19). Understanding Automated Market-Makers, Part 1: Price Impact. Paradigm Research.

<https://research.paradigm.xyz/amm-price-impact>.

Kohli, K. (2020, June 1). How AMMs Work (Explainer Video). DeFi Weekly. <https://defiweekly.substack.com/p/how-amms-work-explainer-video>.

Kohli, K. (2020, June 25). The State of AMMs. DeFi Weekly. <https://defiweekly.substack.com/p/the-state-of-amms-3ad>.

Kozlovski, S. (2021, March 31). Balancer V2-A One-Stop-Shop. Medium. <https://medium.com/balancer-protocol/balancer-v2-a-one-stop-shop-6af1678003f7>.

Krishnamachari, B., Feng, Q., & Grippo, E. (2021). Dynamic Curves for Decentralized Autonomous Cryptocurrency Exchanges.

Martinelli, F. (2021, April 19). Balancer V2: Generalizing AMMs. Medium. <https://medium.com/balancer-protocol/balancer-v2-generalizing-amms-16343c4563ff>.

Naz. (2020, February 24). Crypto Front Running for Dummies. Medium. <https://nazariyv.medium.com/crypto-front-running-for-dummies-bed2d4682db0>.

Pintail. (2020, August 30). Uniswap: A Good Deal for Liquidity Providers? Medium. <https://pintail.medium.com/uniswap-a-good-deal-for-liquidity-providers-104c0b6816f2>.

Powers, B. (2020, December 30). New Research Sheds Light on the Front-Running Bots in Ethereum's Dark Forest. CoinDesk. <https://www.coindesk.com/new-research-sheds-light-front-running-bots-ethereum-dark-forest>.

Understanding Returns on Uniswap. Uniswap Blog RSS. (n.d.).

<https://uniswap.org/docs/v2/advanced-topics/understanding-returns/>.

Uniswap Info DAI/ETH. Uniswap Info. (n.d.).

<https://info.uniswap.org/pair/0xa478c2975ab1ea89e8196811f51a7b7ade33eb11>.

Uniswap. Uniswap Blog RSS. (n.d.). <https://uniswap.org/blog/uniswap-v3/>.

Xu, J., Vavryk, N., Paruch, K., & Cousaert, S. (2021). SoK:

Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols.

Younessi, C. (2019, March 7). Uniswap - A Unique Exchange. Medium.

<https://medium.com/scalar-capital/uniswap-a-unique-exchange-f4cf44f807bf>.

Chapter 4: DEX Aggregators

1inch Network, (2021, March 16). *Introducing the 1inch Aggregation Protocol v3*.

Medium. <https://blog.1inch.io/introducing-the-1inch-aggregation-protocol-v3-b02890986547>.

1inch Network. (2020, December 25). *1INCH token is released*. Medium.

<https://blog.1inch.io/1inch-token-is-released-e69ad69cf3ee>.

Balakrishnan, A. (2021, March 22). *DeFi Aggregators*. Delphi Digital.

<https://www.delphidigital.io/reports/defi-aggregators/>.

Gonella, T. (2021, January 27). *Say hello to 0x v4*. Medium.

<https://blog.0xproject.com/say-hello-to-0x-v4-ce87ca38e3ac>.

Appendix

Hemricourt, P. de. (2020, August 24). *DeFi, DEXes, DEX Aggregators, AMMs, and Built-In DEX Marketplaces, Which is Which and Which is Best?* Medium. <https://medium.com/2key/defi-dexes-dex-aggregators-amms-and-built-in-dex-marketplaces-which-is-which-and-which-is-best-fba04ca48534>.

Kalani, C. (2020, June 30). *Say hello to Matcha!* Matcha. <https://matcha.xyz/blog/say-hello-to-matcha>.

Kalani, C. (2020, October 2). *How Matcha is taking DEX aggregation to a whole new level*. Matcha. <https://matcha.xyz/blog/trading-on-matcha-keeps-getting-better>.

Paraswap. (2021, January 28). *Introducing ParaSwap's new UI & a significant upgrade for our contracts*. Medium.
<https://medium.com/paraswap/introducing-paraswaps-new-ui-a-significant-upgrade-for-our-contracts-ed15d632e1d0>.

Paraswap. (2020, September 17). *Towards a community-owned & driven DeFi middleware*. Medium. https://medium.com/paraswap/towards-a-community-owned-driven-defi-middleware-b7860c55d6a6?source=collection_home---6-----1-----.

Chapter 5: Decentralized Lending & Borrowing

Aave: Open Source DeFi Protocol. Aave. (n.d.). <https://aave.com/>.

Compound Markets. Compound. (n.d.).
<https://compound.finance/markets>

Cream: DeFi Lending Protocol. cream. (n.d.). <https://cream.finance/>.

DeBank | DeFi Wallet for Ethereum Users. DeBank. (n.d.).
<https://debank.com/ranking/lending?chain=eth&date=1Y&select=borrow>.

Lending Protocol Revenue. The Block. (n.d.).

<https://www.theblockcrypto.com/data/decentralized-finance/protocol-revenue>.

Maker Market. Oasis.app. (n.d.). <https://oasis.app/borrow/markets>.

Maker: An Unbiased Global Financial System. MakerDAO. (n.d.).

<https://makerdao.com/en/>.

Protocol Revenue. The Block. (n.d.).

<https://www.theblockcrypto.com/data/decentralized-finance/protocol-revenue>.

Token Terminal Dashboard on Lending Protocols. Token Terminal. (n.d.).

<https://terminal.tokenterminal.com/dashboard/Lending>.

Chapter 6: Decentralized Stablecoins and Stableassets

Tether's Credibility And Its Impact On Bitcoin (Cryptocurrency:BTC-USD).

SeekingAlpha. (n.d.). <https://seekingalpha.com/article/4403640-tethers-credibility-and-impact-on-bitcoin>.

Eva, Matti, Koh, N., & Wangarian. (2021, March 20). *Stability, Elasticity, and Reflexivity: A Deep Dive into Algorithmic Stablecoins*. Deribit Insights.

<https://insights.deribit.com/market-research/stability-elasticity-and-reflexivity-a-deep-dive-into-algorithmic-stablecoins/>.

Empty Set Dollar. (n.d.). *Empty Set Dollar - ESD*. Empty Set Dollar - ESD – Empty Set Dollar. <https://docs.emptyset.finance/>.

Fei Protocol. (2021, January 11). *Introducing Fei Protocol*. Medium.

<https://medium.com/fei-protocol/introducing-fei-protocol-2db79bd7a82b>.

Float Protocol. (2021, March 22). *Announcing Float Protocol and its democratic launch*. Medium. <https://medium.com/float-protocol/announcing-float-protocol-and-its-democratic-launch-d1c27bc21230>.

Float Protocol. (2021, March 22). *FLOAT and the Money Gods*. Medium.

<https://medium.com/float-protocol/float-and-the-money-gods-5509d41c9b3a>.

Frax Finance. *Frax: Fractional-Algorithmic Stablecoin Protocol*. Frax ☒ Finance. (n.d.). <https://docs.frax.finance/>.

Ionescu, S. (2020, October 29). *Introducing Proto RAI*. Medium. <https://medium.com/reflexer-labs/introducing-proto-rai-c4cf1f013ef>.

McKeon, S. (2020, August 12). *The Rise and Fall (and Rise and Fall) of Ampleforth-Part I*. Medium. <https://medium.com/collab-currency/the-rise-and-fall-and-rise-and-fall-of-ampleforth-part-i-cda716dea663>.

Schloss , D., & McKeon, S. (2020, August 12). *The Rise and Fall (and Rise and Fall) of Ampleforth-Part II*. Medium. <https://medium.com/collab-currency/the-rise-and-fall-and-rise-and-fall-of-ampleforth-part-ii-6c0f438e8129>.

Stably. (2021, February 19). *What Uniswap's Liquidity Plunge Reveals about Stablecoins*. Medium. <https://medium.com/stably-blog/what-uniswaps-liquidity-plunge-reveals-about-stablecoins-4fcbee8d210c>.

Watkins, R. (2021, March 3). The Art of Central Banking on Blockchains: Algorithmic Stablecoins. <https://messari.io/article/the-art-of-central-banking-on-blockchains-algorithmic-stablecoins>.

Chapter 7: Decentralized Derivatives

dYdX: Leverage, decentralized. (n.d.) <https://dydx.exchange/>

Hegic: On-chain options trading protocol on Ethereum. (n.d.) <https://www.hegic.co/>

Opyn: Trade Options on Ethereum. (n.d.) <https://www.opyn.co/#/>

Perpetual Protocol: Decentralized Perpetual Contract for every asset. (n.d.)
<https://perp.fi/>.

Perpetual Protocol Exchange. (n.d.) <https://perp.exchange/>.

Synthetix: The Derivatives Liquidity Protocol. (n.d.) <https://synthetix.io/>.

Synthetix Staking Page. (n.d.) <https://staking.synthetix.io/>.

UMA: UMA enables DeFi developers to build synthetic assets. (n.d.)
<https://umaproject.org/>.

UMA KPI Options. (n.d.) <https://claim.umaproject.org/>.

Chapter 8: Decentralized Insurance

Armor.Fi: Smart DeFi Asset Coverage. (n.d.). <https://armor.fi/>.

Cover Protocol: A peer-to-peer coverage market. (n.d.).
<https://www.coverprotocol.com/>.

InsurAce DeFi Insurance. (n.d.). <https://landing.insurace.io/>.

Nexus Mutual Tracker. (n.d.). <https://nexustracker.io/>.

Nexus Mutual: A decentralised alternative to insurance. (n.d.).
<https://nexusmutual.io/>.

Nsure.Network: Experience DeFi, Risk Free. (n.d.).
<https://nsure.network/#/home>.

Unslashed Finance: The Insurance Products that crypto needs. (n.d.).
<https://www.unslashed.finance/>.

Chapter 9: Decentralized Indices

Campbell, L. (2020, November 11). The best DeFi indices for your crypto portfolio. Bankless. <https://newsletter.banklesshq.com/p/the-best-defi-indices-for-your-crypto>.

Dune Analytics. (n.d.). <https://duneanalytics.com/0xBoxer/indices-products>.

Governance on Index Coop. Snapshot. (n.d.).
<https://snapshot.org/#/index>.

Governance on Indexed Finance. Snapshot. (n.d.).
<https://gov.indexed.finance/#/ndx.eth/proposal/QmdVmMefXUAUqU1xgfjeUic4o4Ud4cqFKwyABaQSBnQNG9>.

Inc., M. S. C. I. (2010). Update on Msci Equal Weighted Indices. SSRN Electronic Journal. <https://doi.org/10.2139/ssrn.1729770>.

Index Cooperative Website. Index. (n.d.). <https://www.indexcoop.com/>.

Indexed Finance Official Documentation. Indexed Finance. (n.d.).
<https://docs.indexed.finance/>.

Indexed Finance Official Website. Indexed. (n.d.).
<https://indexed.finance/>.

McKee, S. L. (2016, July 18). Cap Weighted Versus Equal Weighted, Which Approach Is Better? Forbes.
<https://www.forbes.com/sites/investor/2016/07/18/cap-weighted-versus-equal-weighted-which-approach-is-better/?sh=727e63e847c3>.

PowerPool Official Blog. Medium. (n.d.).
<https://medium.com/@powerpoolcvp>.

PowerPool Official Website. PowerPool. (n.d.).
<https://powerpool.finance/>.

PowerPool. (2021, January 24). PowerIndex v2: Unlimited ETFs & Automated Portfolio Strategies. Medium.

<https://medium.com/powerpool/powerindex-v2-unlimited-etfs-automated-portfolio-strategies-6086917e6348>.

pr0, G2theM, Norsefire, Tai, John, RickieJean, ... Noice. (2021, January 30). Oracle Top 5 Token Index proposal. Indexed Finance.

<https://forum.indexed.finance/t/oracle-top-5-token-index-proposal/89>.

Chapter 10: Decentralized Prediction Markets

Augur. *The Ultimate Guide to Decentralized Prediction Markets*. Augur. (n.d.).

<https://augur.net/blog/prediction-markets>.

Beneš, N. (2018, April 6). *How manipulation-resistant are Prediction Markets?* Medium. <https://blog.gnosis.pm/how-manipulation-resistant-are-prediction-markets-710e14033d62>.

Fletcher-Hill, P. (2019, February 7). *A guide to Augur market economics*. Medium. <https://medium.com/veil-blog/a-guide-to-augur-market-economics-16c66d956b6c>.

Gnosis. (2020, July 5). *Omen and the Next Generation of Prediction Markets*. Medium. <https://blog.gnosis.pm/omen-and-the-next-generation-of-prediction-markets-2e7a2dd604e>.

Omen Prediction Markets. Omen. (n.d.). <https://omen.eth.link/>.

The World's Most Accessible, No-Limit Betting Platform. Augur. (n.d.).

<https://augur.net/>.

Chapter 11: Decentralized Fixed-Interest Rate Protocols

Horizon. (n.d.). <https://horizon.finance/#/>.

How Tranche Lending Will Bring Fixed Interest Rates to DeFi. ConsenSys. (2021, February 4). <https://consensys.net/blog/codefi/how-tranche-lending-will-bring-fixed-interest-rates-to-defi/>.

Introduction. Introduction · GitBook. (n.d.). <https://docs.yield.is/>.

Rai, R. (n.d.). *Fixed Income Protocols: The Next Wave of DeFi Innovation.* Messari Crypto News. <https://messari.io/article/fixed-income-protocols-the-next-wave-of-defi-innovation>.

saffron.finance. Saffron. (n.d.). <https://saffron.finance/>.

Woodward, T. (2020, November 19). *Why Fixed Rates Matter.* Medium. <https://medium.com/notional-finance/why-fixed-rates-matter-1b03991275d6>.

Chapter 12: Decentralized Yield Aggregators

Alpha Finance Lab. (2021, April 19). <https://alphafinance.io/>.

Badger Finance: Community Rules Everything. (2020, December 27). <https://badger.finance/>.

Defi Lego connects as Yearn Finance announces five mergers in a week. Brave New Coin. (n.d.). <https://bravenewcoin.com/insights/defi-lego-connects-as-yearn-finance-announces-five-mergers-in-a-week>.

Harvest Finance. (n.d.). <https://harvest.finance/>.

yearn.finance. (n.d.). <https://yearn.finance/>.

Yearn.finance. (2021, March 5). We have decided to end the previously announced merger process of Yearn and Cover. Both protocols will continue to operate independently. yVault depositors who have previously purchased Cover protection are unaffected by this. Twitter. <https://twitter.com/iearnfinance/status/1367796331507552258>.

Yearn Finance Launches v2 Vaults, YFI Token Jumps 15%. BeInCrypto. <https://beincrypto.com/yearn-finance-v2-vaults-yfi-token/>.

Chapter 13: Oracles and Data Aggregators

Band Protocol: Secure, Scalable Blockchain-Agnostic Decentralized Oracle. (n.d.) <https://bandprotocol.com/>.

BandChain. (n.d.) <https://bandprotocol.com/bandchain>.

Chainlink: Connect your smart contract to the outside world. (n.d.) <https://chain.link/>.

Chainlink. (2021, April 30). Chainlink 2.0 Lays Foundation for Adoption of Hybrid Smart Contracts. Chainlink. <https://blog.chain.link/chainlink-2-0-lays-foundation-for-adoption-of-hybrid-smart-contracts/>.

Covalent: One unified API. One billion possibilities. (n.d.). <https://www.covalenthq.com/>.

The Graph: APIs for a vibrant decentralized future. (n.d.) <https://thegraph.com/>.

Chapter 14: Multi-Chain Protocols & Cross-Chain Bridges

An Introduction to Binance Bridge. (n.d.) <https://academy.binance.com/en/articles/an-Introduction-to-binance-bridge>.

AnySwap Dashboard. (n.d.) <https://anyswap.exchange/dashboard>.

Binance Bridge. (n.d.) <https://www.binance.org/en/bridge>.

Chainlist: Helping Users Connect to EVM powered networks. (n.d.)
<https://chainlist.org/>.

Documentation on Terra Bridge. (n.d.) <https://docs.mirror.finance/user-guide/terra-bridge>.

Terra Bridge. (n.d.) <https://bridge.terra.money/>.

ThorChain Technology. (n.d.). <https://thorchain.org/technology#how-does-it-work>.

Chapter 15: DeFi Exploits

Etherscan Information Center. (n.d.). <https://info.etherscan.com/>.

Flash Loans. Aave FAQ. (n.d.). <https://docs.aave.com/faq/flash-loans>.

Furucombo: Create all kinds of DeFi combo. (n.d.).
<https://furucombo.app/>.

ImmuneFi: DeFi's leading bug bounty platform. (n.d.).
<https://immunefi.com/>.

rekt. (n.d.). <https://www.rekt.news/>.

Chapter 16: DeFi will be the New Normal

Bambysheva, N. (2021, March 29). Visa Will Start Settling Transactions With Crypto Partners In USDC On Ethereum. Forbes.
<https://www.forbes.com/sites/ninabambysheva/2021/03/29/visa-to-start-settling-transactions-with-bitcoin-partners-in-usdc>.

Bitcoin Headlines. (2021, March 21) Documentation Bitcoin:
<https://twitter.com/DocumentingBTC/status/1372919635083923460>

Buterin, V. (2020, December 28). *Endnotes on 2020: Crypto and Beyond.*
[https://vitalik.ca/general/2020/12/28/endnotes.html.](https://vitalik.ca/general/2020/12/28/endnotes.html)

Cronje, A. (2021, January 12). *Building in defi sucks (part 2).* Medium.
[https://andrecronje.medium.com/building-in-defi-sucks-part-2-75df9ee7871b.](https://andrecronje.medium.com/building-in-defi-sucks-part-2-75df9ee7871b)

stani.eth (2021, May 17). Aave Pro for institutions
pic.twitter.com/sUWOFDWcx. Twitter.
[https://twitter.com/StaniKulechov/status/1394390461968633859.](https://twitter.com/StaniKulechov/status/1394390461968633859)

Thurman, A. (2021, May 12). DeFi lending platform Aave reveals
'permissioned pool' for institutions. Cointelegraph.
[https://cointelegraph.com/news/defi-lending-platform-aave-reveals-private-pool-for-institutions.](https://cointelegraph.com/news/defi-lending-platform-aave-reveals-private-pool-for-institutions)

GLOSSARY

Index	Term	Description
A	Airdrop	Airdrop refers to the distribution of a reserve of tokens, usually to users who have completed certain actions or fulfill certain criterias.
	Annual Percentage Yield (APY)	It is an annualized return on saving or investment and the interest is compounded based on the period.
	Admin Key Risk	It refers to the risk where the master private key for the protocol could be compromised.
	Algorithmic Stablecoins	Algorithmic stablecoins utilize algorithms to control the stablecoin's market structure and the underlying economics.
	Algorithmic Stableassets	Unlike algorithmic stablecoins, algorithmic stableassets could be seen as another form of collateral rather than units of accounts

Index	Term	Description
	Automated Market Maker (AMM)	Automated Market Maker removes the need for a human to manually quote bids and ask prices in an order book and replaces it with an algorithm.
	Audit	Auditing is a systematic process of examining an organization's records to ensure fair and accurate information the organization claims to represent. Smart contract audit refers to the practice of reviewing the smart contract code to find vulnerabilities so that they can be fixed before it is exploited by hackers.
	An Application Programming Interface (API)	An interface that acts as a bridge that allows two applications to interact with each other. For example, you can use CoinGecko's API to fetch the current market price of cryptocurrencies on your website.
B	Back-running	It is the act of when the attacker sells the tokens right after front-running the victim's trade to make a risk-free arbitrage. See front-running and sandwich attack.
	Buy and Hold	This refers to a TokenSets trading strategy which realigns to its target allocation to prevent overexposure to one coin and spreads risk over multiple tokens.
	Bridge	A protocol that connects two blockchains together, allowing users to transfer assets between them.

Index	Term	Description
	Bonding Curve	A bonding curve is a mathematical curve that defines a dynamic relationship between price and token supply. Bonding curves act as an automated market maker where as the number of supply of a token decreases, the price of the token increases. It is useful as it helps buyers and sellers to access an instant market without the need of intermediaries.
C	Cryptocurrency Exchange (Cryptoexchange)	It is a digital exchange that helps users exchange cryptocurrencies. For some exchanges, they also facilitate users to trade fiat currencies to cryptocurrencies.
	Custodian	Custodian refers to the third party to have control over your assets.
	Fiat-collateralized stablecoin	A stablecoin that is backed by fiat-currency. For example, 1 Tether is pegged to \$1.
	Crypto-collateralized stablecoin.	A stablecoin that is backed by another cryptocurrency. For example, Dai is backed by Ether at an agreed collateral ratio.
	Centralized Exchange (CEX)	Centralized Exchange (CEX) is an exchange that operates in a centralized manner and requires full custody of users' funds.
	Collateral	Collateral is an asset you will have to lock-in with the lender in order to borrow another asset. It acts as a guarantor that you will repay your loan.

Index	Term	Description
	Collateral Ratio	Collateral ratio refers to the maximum amount of asset that you can borrow after putting collateral into a DeFi decentralized application.
	cTokens	cTokens are proof of certificates that you have supplied tokens to Compound's liquidity pool.
	Cryptoasset	Cryptoasset refers to digital assets on blockchain. Cryptoassets and cryptocurrencies generally refer to the same thing.
	Cover Amount	It refers to the maximum payable money by the insurance company when a claim is made.
	Claim Assessment process	It is the obligation by the insurer to review the claim filed by an insurer. After the process, the insurance company will reimburse the money back to the insured based on the Cover Amount.
	Composability	Composability is a system design principle that enables applications to be created from component parts.
	Cross-chain	Transactions that occur between different blockchains.
D	Decentralized Finance (DeFi)	DeFi is an ecosystem that allows for the utilization of financial services such as borrowing, lending, trading, getting access to insurance, and more without the need to rely on a centralized entity.

Index	Term	Description
	Decentralized Applications (Dapps)	Applications that run on decentralized peer-to-peer networks such as Ethereum.
	Decentralized Autonomous Organization (DAO)	Decentralized Autonomous Organizations are rules encoded by smart contracts on the blockchain. The rules and dealings of the DAO are transparent and the DAO is controlled by token holders.
	Decentralized Exchange (DEX)	Decentralized Exchange (DEX) allows for trading and direct swapping of tokens without the need to use a centralized exchange.
	Derivatives	Derivative comes from the word derive because it is a contract that derives its value from an underlying entity/product. Some of the underlying assets can be commodities, currencies, bonds, or cryptocurrencies.
	Dai Saving Rate (DSR)	The Dai Savings Rate (DSR) is an interest earned by holding Dai over time. It also acts as a monetary tool to influence the demand of Dai.
	Dashboard	A dashboard is a simple platform that aggregates all your DeFi activities in one place. It is a useful tool to visualize and track where your assets are across the different DeFi protocols.
	Data Aggregator	Service providers that index and aggregate data so that it be queried by other decentralized applications

Index	Term	Description
E	Ethereum	Ethereum is an open-source, programmable, decentralized platform built on blockchain technology. Compared to Bitcoin, Ethereum allows for scripting languages which has allowed for application development.
	Ether	Ether is the cryptocurrency that powers the Ethereum blockchain. It is the fuel for the apps on the decentralized Ethereum network
	ERC-20	ERC is an abbreviation for Ethereum Request for Comment and 20 is the proposal identifier. It is an official protocol for proposing improvements to the Ethereum network. ERC-20 refers to the commonly adopted standard used to create tokens on Ethereum.
	Exposure	Exposure refers to how much you are 'exposed' to the potential risk of losing your investment. For example, price exposure refers to the potential risk you will face in losing your investment when the price moves.
F	Future Contract	It is a contract which you enter to buy or sell a particular asset at a certain price at a certain date in the future.
	Factory Contract	It is a smart contract that is able to produce other new smart contracts.
	Fixed-Interest Rate Protocol (FIRP)	A new class of protocols that have a fixed interest rate element.

Glossary

Index	Term	Description
	Flash Loans	Borrowers can take up loans with zero collateral if the borrower repays the loan and any additional interest and fees within the same transaction.
	Front-Running	In the cryptocurrency context, frontrunning works in DEX where orders made are broadcasted to the blockchain for all to see, a frontrunner will attempt to listen to the blockchain to pick up suitable orders to frontrun by orders on the market and placing enough fees to have the transaction mined faster than the target's orders.
	Funding Rate	Periodic payments made by traders based on the difference between the perpetual contract prices and spot prices of an asset.
G	Gas	Gas refers to the unit of measure on the amount of computational effort required to execute a smart contract operation on Ethereum.
	Governance	To steer the direction of the DeFi protocol, governance is introduced whereby the project community can decide collectively. To make this possible, governance tokens are pioneered by Compound, allowing token holders to vote on protocol proposals that any community member can submit.

Index	Term	Description
H	Hard Fork	Forced bifurcation of a blockchain, which is usually given when a fairly significant change is implemented in the software code of a network. It results in a permanent divergence of a blockchain into two blockchains. The original blockchain does not recognize the new version.
I	IDO	IDO stands for Initial Decentralized Exchange Offering or Initial DEX offering. This is where tokens are first offered for sale to the public using a DEXs liquidity pool.
	IBCO	IBCO stands for Initial Bonding Curve Offering. Token prices are based on a curve, where subsequent investors will push up the token's price.
	IFO	IFO stands for Initial Farm Offering. Typically, users stake their assets in exchange for the project's tokens. The project then receives the user's staked assets as payment.
	IMAP	IMAP stands for Internet Message Access Protocol. It is an Internet protocol that allows email applications to access email on TCP/IP servers.
	Impermanent Loss	Temporary loss of funds due to volatility leading to divergence in price between token pairs provided by liquidity providers.
	Index	An index measures the performance of a basket of underlying assets. An index moves when the overall performance of the underlying assets in the basket moves.

Glossary

Index	Term	Description
	Insurance	An agreement to provide compensation for losses incurred in exchange for upfront payment.
	Inverse	This Synthetix strategy is meant for those who wish to “short” a benchmark. Traders can purchase this when they think a benchmark is due to decrease.
J		-
K	Know-Your-Customer (KYC)	Know-Your-Customer (KYC) is a compliance process for business entities to verify and assess their clients.
L	Layer-1 Chains	Blockchains where every transaction is settled and verified on the network itself.
	Layer-2 Chains	Layer 2 is a chain that is built on top of the base chain to improve scalability without compromising the security and the decentralization.
	Liquidation Penalty	It is a fee that a borrower has to pay along with their liquidated collateral when the value of their collateral asset falls below the minimum collateral value.
	Liquidation Ratio	The ratio of collateral to debt at which your collateral will be subject to liquidation if it falls below it.
	Liquidity Bootstrapping Pool	Liquidity pools where projects can sell tokens via a configurable AMM. They are mainly used to improve price discovery and reduce volatility.

Index	Term	Description
	Liquidity Pools	Liquidity pools are token reserves that sit on smart contracts and are available for users to exchange tokens. Currently the pools are mainly used for swapping, borrowing, lending, and insurance.
	Liquidity Mining	The reward program of giving out the protocol's native tokens in exchange for capital. It is a novel way to attract the right kind of community participation for DeFi protocols.
	Liquidity Risk	A risk when protocols like Compound could run out of liquidity.
	Liquidity Providers	Liquidity providers are people who loan their assets into the liquidity pool. The liquidity pool will increase as there are more tokens.
	Liquidity Pool Aggregator	It is a system which aggregates liquidity pools from different exchanges and is able to see all available exchange rates in one place. It allows you to compare for the best possible rate.
	Leverage	It is an investment strategy to gain higher potential return of the investment by using borrowed money.
M	MakerDAO	MakerDAO is the creator of Maker Platform and DAO stands for Decentralized Autonomous Organisation. MakerDAO's native token is MKR and it is the protocol behind the stablecoins, SAI and DAI.

Index	Term	Description
	Market Maker Mechanisms	A Market Maker Mechanism is an algorithm that uses a bonding curve to quote both a buy and a sell price. In the crypto space, Market Maker Mechanism is mainly used by Uniswap or Kyber to swap tokens.
	Margin Trading	It is a way of investing by borrowing money from a broker to trade. In DeFi, the borrowing requires you to collateralize assets.
	MKR	Maker's governance token. Users can use it to vote for improvement proposal(s) on the Maker Decentralized Autonomous Organization (DAO).
	Mint	It refers to the process of issuing new coins/tokens.
	Multichain	Usually refers to products or tokens that exist on one or more blockchains.
N	Node	Within the blockchain network, the nodes are computers that connect to the network and have an updated copy of the blockchain. Together with the miners they are the guarantors that the network works properly. The nodes in Bitcoin are very important because they help the mission of keeping the network decentralized.
O	Order book	It refers to the list of buying and selling orders for a specific asset at various price levels.

Index	Term	Description
	Over-collateralization	Over-collateralization refers to the value of a collateral asset that must be higher than the value of the borrowed asset.
	Option	Option is a right but not the obligation for someone to buy or sell a particular asset at an agreed price on or before an expiry date.
	Oracle	Service providers which collect and verify off-chain data to be provided to smart contracts on the blockchain.
P	Perpetual Swaps	Enable users to essentially open a leveraged position on a futures contract with no expiration date.
	Prediction Markets	Prediction markets are markets created for participants to bet on the outcomes of future events.
	Price discovery	Price discovery refers to the act of determining the proper price of an asset through several factors such as market demand and supply.
	Protocol	A protocol is a base layer of codes that tells something on how to function. For example, Bitcoin and Ethereum blockchains have different protocols.

Glossary

Index	Term	Description
	Peer-to-Peer	In blockchain, "peer" refers to a computer system or nodes on a decentralized network. Peer-to-Peer (P2P) is a network where each node has an equal permission to validating data and it allows two individuals to interact directly with each other.
	Perpetuals	It refers to perpetual futures, which is an agreement to purchase or sell an asset in the future without a specified date.
Q		-
R	Range Bound	This TokenSets strategy automates buying and selling within a designated range and is only intended for bearish or neutral markets.
	Rebalance	It is a process of maintaining a desired asset allocation of a portfolio by buying and selling assets in the portfolio.
	Risk Assessor	Someone who stakes value against smart contracts in Nexus Mutual. He/she is incentivized to do so to earn rewards in NXM token, as other users buy insurance on the staked smart contracts.

Index	Term	Description
	Rug-Pull	In the context of crypto and Decentralized Finance (DeFi), having been rug pulled means to have buy support or Decentralized Exchange (DEX) liquidity pool taken away from a market. This results in a sell death spiral as other liquidity providers, holders and traders sell to salvage their holdings. Typically, it is a new form of exit scamming where someone will drain the pool at DEX, leaving the token holders unable to trade.
S	Sandwich Attack	It is a combination attack of front-running and back-running. Thus, the attacker ‘sandwiches’ the victim’s trade to make a risk-free arbitrage.
	Smart Contracts	A smart contract is a programmable contract that allows two counterparties to set conditions of a transaction without needing to trust another third party for the execution.
	Stablecoins	A stablecoin is a cryptocurrency that is pegged to another stableasset such as the US Dollar.
	Staking	Staking can mean various things in crypto space. Generally, staking refers to locking up your cryptoassets in a dApp. Otherwise, it could also refer to participation in a Proof-of-Stake (PoS) system to put your tokens in to serve as a validator to the blockchain and receive rewards.

Index	Term	Description
	Soft Fork	Due to the decentralized nature of the peer-to-peer cryptocurrency network, any updates or changes must be agreed by all participating nodes. Such code changes in the blockchain occur via chain forks, whereby when the network virtually splits in 2, each following different sets of rules. A soft fork event refers to when a fork occurred but old nodes can still participate in the network.
	Spot market	Spot market is the buying and selling of assets with immediate delivery.
	Speculative activity	It is an act of buying and selling, while holding an expectation to gain profit.
	Spread Surplus	The net positive difference between swap transactions when the executed price is slightly better than the price quoted
	Stability Fee	It is equivalent to the ‘interest rate’ which you are required to pay along with the principal debt of the vault.
	Slippage	Slippage is the difference between the expected price and the actual price where an order was filled. It is generally caused by low liquidity.
	Synths	Synths stand for Synthetic Assets. A Synth is an asset or mixture of assets that has/have the same value or effect as another asset.

Index	Term	Description
	Smart Contract Cover	An insurance offer from Nexus Mutual to protect users against hacks in smart contracts that store value.
T	TCP/IP	It stands for Transmission Control Protocol/Internet Protocol. It is a communication protocol to interconnect network devices on the internet.
	Total Value Locked	Total Value Locked refers to the cumulative collateral of all DeFi products.
	Tokens	It is a unit of a digital asset. Token often refers to coins that are issued on existing blockchain.
	Tokenize	It refers to the process of converting things into digital tradable assets
	Technical Risk	It refers to the bugs on smart contracts which can be exploited by hackers and cause unintended consequences.
	Trading Pairs	A trading pair is a base asset that is paired with its target asset in the trading market. For example, for the ETH/DAI trading pair, the base asset is ETH and its target pair is DAI.
	Trend Trading	This strategy uses Technical Analysis indicators to shift from 100% target asset to 100% stableasset based on the implemented strategy.

Glossary

Index	Term	Description
U	Utilization Ratio	Utilization ratio is a common metric that has been used in legacy finance where you are measuring how much you borrow against your borrowing limit. Similarly, we can calculate decentralized lending application utilization ratio by measuring the borrowing volume against the value locked within the lending dApp.
V	Value Staked	It refers to how much value the insurer will put up against the target risk. If the value that the insurer staked is lower than the target risk, then it is not coverable.
	Validators	In contrast to mining on a Proof of Work blockchain network, Proof of Stake blockchain networks are secured by a distributed consensus of dedicated validators who have staked (locked into the network) a significant amount of token as long as the validator nodes are running. Validators are queued for block-signing based on a combination of random selection, amount (weight) staked and length of time staked (age) and others depending on the design of the consensus algorithm.
W	Wallet	A wallet is a user-friendly interface to the blockchain network that can be used as a storage, transaction and interaction bridge between the user and the blockchain.

Index	Term	Description
	Wrapped Asset	Represent assets that exist on other networks besides their native blockchain. For example, WBTC is the ERC-20 version of Bitcoin which exists on the Ethereum blockchain.
X		-
Y	Yield Farming	It refers to the act of staking or lending digital assets in order to generate a return, usually in the form of other tokens.
	Yield Aggregator	Yield aggregators are born to serve the need of automating users' investment strategies, sparing them the trouble of monitoring the market for the best yield farms.
Z	ZK Rollup	A Layer-2 scaling solution where transfers are bundled together and executed in one transaction on the base chain. Among these solutions are Starkware and Loopring.

In collaboration with
the Wharton Blockchain and Digital Asset Project



Decentralized Finance (DeFi) Policy-Maker Toolkit

WHITE PAPER
JUNE 2021

Contents

3	Foreword
4	Executive summary
5	1 What is DeFi?
6	1.1 Distinguishing characteristics
7	1.2 The DeFi architecture
10	1.3 DeFi service categories
12	2 Risks
14	2.1 Financial
15	2.2 Technical
16	2.3 Operational
18	2.4 Legal compliance
18	2.5 Emergent risks
20	3 Policy approaches
21	3.1 DeFi and financial regulation
22	3.2 Available policy tools
24	3.3 Decision tree
25	Conclusion
26	Appendix 1: Background assessment
28	Appendix 2: Stakeholder mapping tool
30	Appendix 3: Decentralization spectrum
31	Appendix 4: DeFi policy-maker canvas
34	Contributors
35	Acknowledgements
36	Endnotes

© 2021 World Economic Forum. All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, or by any information storage and retrieval system.

Foreword



Sumedha Deshmukh
Platform Curator,
Blockchain and Digital Assets,
World Economic Forum



Sheila Warren
Deputy Head, Centre
for the Fourth Industrial
Revolution; Member of
the Executive Committee,
World Economic Forum



Kevin Werbach
Professor of Legal Studies &
Business Ethics, and Director,
Blockchain and Digital Asset
Project, Wharton School,
University of Pennsylvania

Decentralized finance (DeFi) is an emerging and rapidly evolving area in the blockchain environment. Although examples of DeFi have existed for several years, there was a sudden upsurge of activity in 2020. In one year, the value of digital assets¹ locked in DeFi smart contracts grew by a factor of 18, from \$670 million to \$13 billion; the number of associated user wallets grew by a factor of 11, from 100,000 to 1.2 million; and the number of DeFi-related applications grew from 8 to more than 200.² This growth in turn has stimulated interest from both the private and public sectors.

DeFi aims to reconstruct and reimagine financial services on the foundations of distributed ledger technology, digital assets and smart contracts. As such, DeFi is a noteworthy sector of financial technology (fintech) activity.

However, serious questions remain:

- What, if any, are the distinctive aspects of DeFi? What distinguishes a DeFi service from a similar service based on traditional finance?
- What are the opportunities and potential benefits of DeFi? To whom will these benefits accrue – and who might be excluded or left behind?
- What are the risks – individual, organizational and systemic – of using DeFi? How do these risks apply to clients, markets, counterparties and beyond?

- Can DeFi become a significant alternative to traditional financial services? If so, will there be points of integration? If not, what if anything will DeFi represent in the market?
- What novel legal and policy questions does DeFi raise? How should policy-makers approach DeFi? What options exist for addressing these questions?

Notably, the DeFi space is relatively nascent and rapidly evolving, so the full scope of risks and potential for innovation remain to be seen – and there are unique challenges in regulating and creating policies for such a new and changing area. This report does not recommend any one single approach; instead, it is designed as a set of tools that can be applied in light of the legal contexts and policy positions of each jurisdiction, which may vary. In the appendices we offer a series of worksheets and other tools to assist with the evaluation of DeFi activities. A companion piece, [DeFi Beyond the Hype](#), provides additional detail about the major DeFi service categories.

Our hope is that this resource will enable regulators and policy-makers to develop thoughtful approaches to DeFi, while helping industry participants understand and appreciate public-sector concerns. It is the result of an international collaboration among academics, legal practitioners, DeFi entrepreneurs, technologists and regulatory experts. It provides a solid foundation for understanding the major factors that should drive policy-making decisions.

Executive summary

Decentralized finance (“DeFi”) is a broad term for financial services that build on top of the decentralized foundations of blockchain technology. The space has evolved since the 2015 launch of the Ethereum network, which laid the groundwork by implementing blockchain-based smart contracts.³ There has been increased interest recently, paralleling the 2013 spike in bitcoin price and the 2017 boom in initial coin offerings.⁴ As new DeFi services aspire to reinvent elements of financial services, and billions of dollars of digital assets are pledged to DeFi capital pools, policy-makers and regulators face significant challenges in balancing its risks and opportunities.

DeFi proponents say it can address challenges within the traditional financial system.^{5,6} Open-source technology, economic rewards, programmable smart contracts and decentralized governance might offer greater efficiencies, opportunities for inclusion, rapid innovation and entirely new financial service arrangements.⁷ On the other hand, DeFi raises considerations related to consumer protection, loss of funds, governance complexities, technical risk and systemic risk. Significant incidents involving technical failures and attacks on DeFi services have already occurred.⁸ Moreover, questions remain about the actual extent of decentralization of some protocols – and associated risks, e.g. for manipulation – and whether DeFi is more than a risky new vehicle for speculation that may open the door to fraud and illicit activity.⁹

The purpose of this document is to highlight DeFi’s distinguishing characteristics and opportunities while also calling attention to new and existing risks – including the scope, significance and challenges of the fast-growing DeFi ecosystem. Understanding DeFi business models and the full set of relationships underlying DeFi is crucial for an accurate risk assessment and nuanced policy-making.

This toolkit:

- Provides an overview of the DeFi space generally, and the major classes of DeFi protocols, with tools to help understand the implications of new services
- Explores the potential benefits of the DeFi approach, along with the challenges that DeFi businesses will face
- Offers a detailed breakdown of the risks that DeFi may pose. Many of these are familiar concerns (although sometimes manifested differently), while others are unique to the decentralized, programmable and composable structure of DeFi
- Maps out potential legal and regulatory responses to DeFi

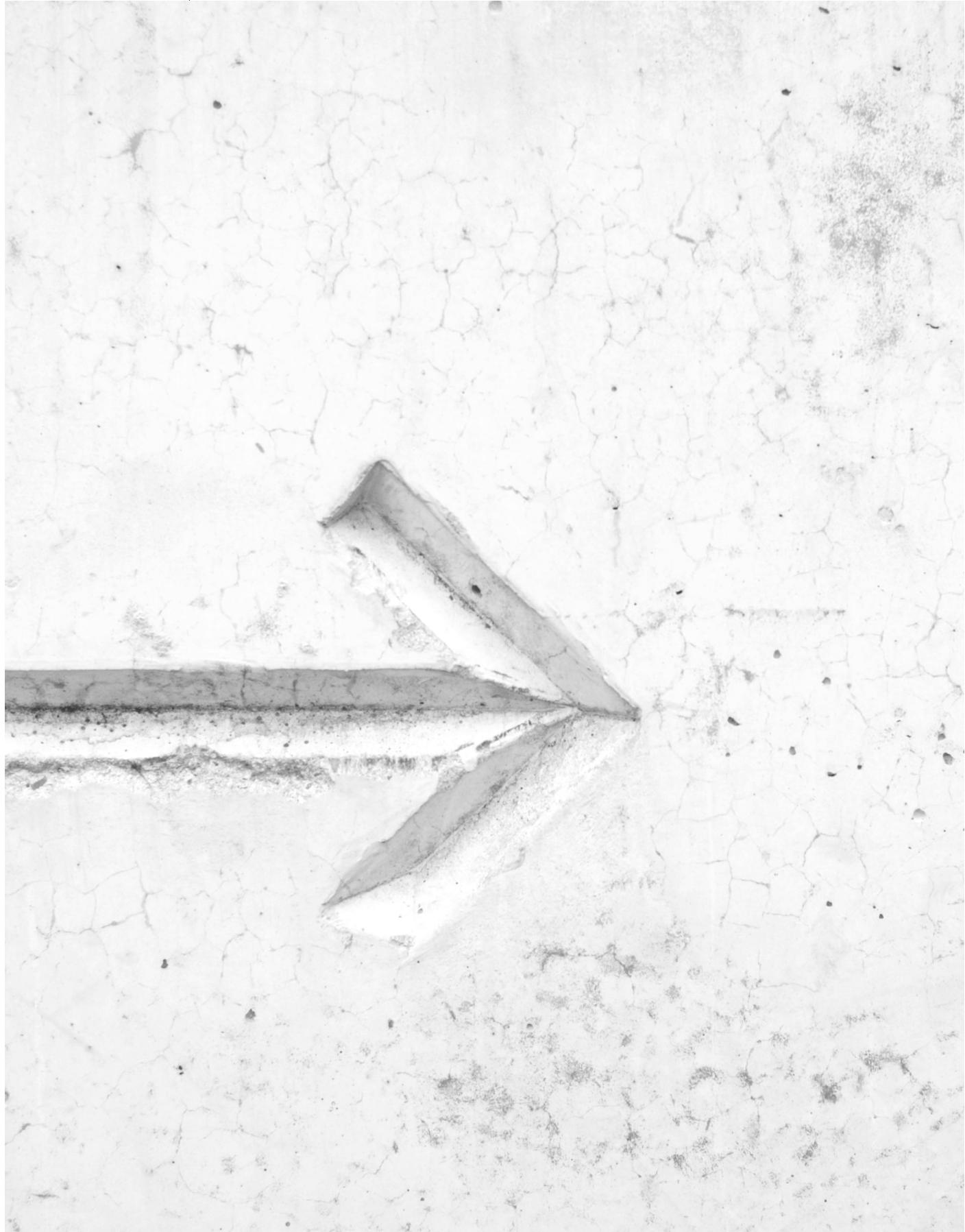
Our goal is not to recommend any specific actions universally, but to identify potential approaches and important considerations for the DeFi context. Financial regulatory regimes vary from jurisdiction to jurisdiction, as do policy-makers’ judgements about the relative risks and rewards. DeFi will raise further questions about whether regulators have the proper tools to address evolving market activity, and how they can assert jurisdiction over a set of technologies and stakeholders that is intrinsically borderless and global.

Appendix 1 offers a background assessment for policy-makers and regulators looking to understand whether DeFi may be relevant to their entity.

Appendix 2 provides a stakeholder mapping tool for DeFi services. **Appendix 3** outlines the decentralization spectrum, while **Appendix 4** provides a DeFi policy-maker canvas.

1

What is DeFi?



“DeFi” is a general term for an evolving trend. Broadly, it is a category of blockchain-based decentralized applications (DApps) for financial services. DeFi encompasses a variety of technologies, business models and organizational structures,¹⁰ generally replacing traditional forms of intermediation. DeFi transactions involve digital assets and generally operate on top of base-layer settlement platforms.

- **DeFi protocols** define software specifications and interfaces to create, manage and convert digital assets, building on a blockchain settlement layer.

– **DeFi services** implement DeFi protocols to create financial services, and associated functions such as specification of risk parameters and interest rates.¹¹

- **DeFi users** access DeFi services to transact.

DeFi services may be made available to users through centralized web applications or permissionless interfaces such as programmable wallets or smart contracts. They may be provided by a traditional controlling entity, a community around a non-profit entity or a *decentralized autonomous organization (DAO)*, in which rights and obligations are specified in smart contracts.

1.1 | Distinguishing characteristics

While the space is evolving quickly, we offer a functional description to distinguish DeFi from traditional financial services and auxiliary services. A DeFi protocol, service or business model has the following four characteristics:

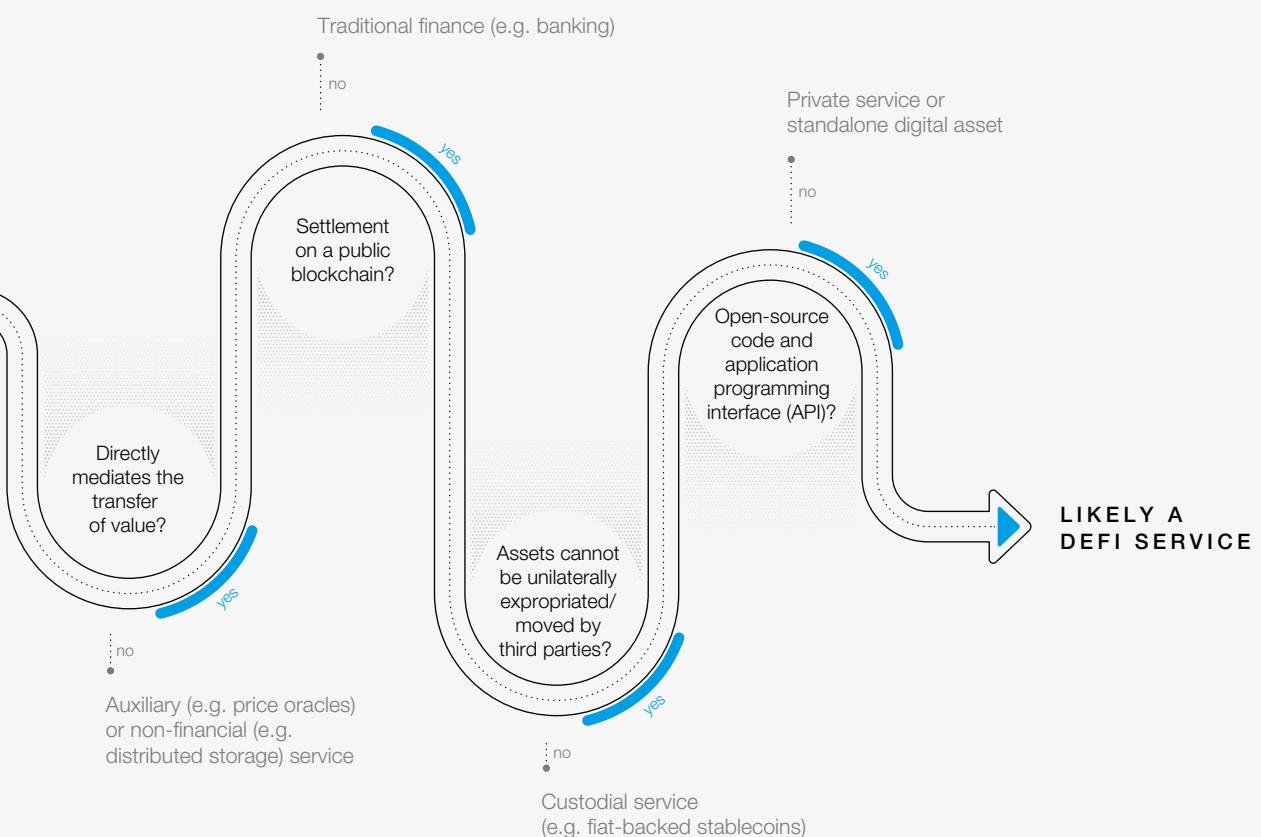
-  **1. Financial services or products**
-  **2. Trust-minimized operation and settlement**
-  **3. Non-custodial design**
-  **4. Programmable, open and composable architecture**

Importantly, these characteristics represent the aspirations for DeFi. Businesses will exhibit each of these characteristics to varying degrees, and this may be fluid over projects’ lifetimes.¹² Broadly speaking, the goal of DeFi solutions is to provide functions analogous to, and potentially beyond, those offered by traditional financial service providers, without reliance on central intermediaries or institutions.

Figure 1 provides a flow chart for evaluating whether an offering should be classified as DeFi.

FIGURE 1

DeFi classification flow chart



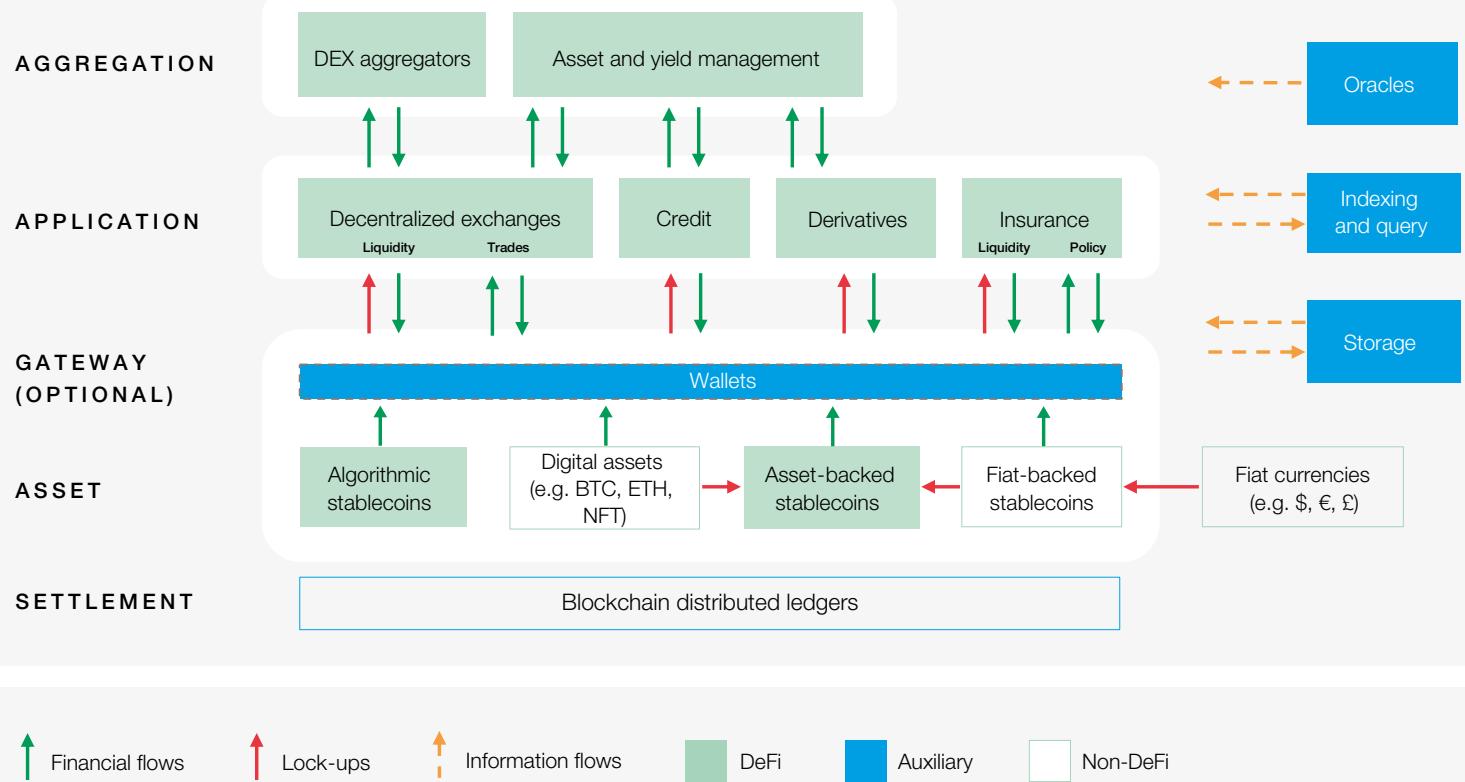
1.  **Financial services or products** means processing or directly enabling the transfer of value among parties. They are distinguished from information services, such as price feeds or storage, that only indirectly support value transfer.
 2.  **Trust-minimized operation and settlement** means that transactions are executed and recorded according to the explicit logic of a DeFi protocol's predetermined rules, on a permissionless basis. That is, due to their availability through a decentralized settlement layer, transactions do not require trust in the counterparty or a third-party intermediary. While the platforms vary, DeFi projects generally build on public, permissionless blockchains.¹³ To date, most activity has been on the Ethereum blockchain, but activity is growing on other networks such as Algorand, Avalanche, Binance Smart Chain, Cosmos, EOS, NEAR, Polkadot, Solana and Tezos.¹⁴
 3.  **Non-custodial design** means that the assets issued or managed by DeFi services cannot be unilaterally expropriated or altered by parties other than the account owner, even those providing intermediation and other services.¹⁵ These tokens are subject only to the explicit logic of their smart contracts and the relevant DeFi protocols. Changes in those protocols, executed through the relevant governance structures, may affect the economic rights of digital asset holders.
 4.  **Open, programmable and composable architecture** means that there is broad availability of the underlying source code for DeFi protocols and a public application programming interface (API) enabling service composability, similar to open banking¹⁶ for centralized financial services. The widespread use of *open-source code* allows participants to view and verify protocols directly, and to fork code – take source code and create an independent development – or to create derivative or competitive services. The use of *open interfaces* means that third parties can understand, extend and verify the integrity and security of the service. Together with the API, this enables access to functionality in an automated, permissionless way. It also allows for *programmability*: customizing and extending financial instruments dynamically. For example, the terms of a derivative may be specified at the time of its creation, and then enforced immutably through the decentralized settlement layer.
- Composability* means that these programmatic components can be combined to create financial instruments and services, including those incorporating multiple DeFi services and protocols. For example, a stablecoin may be used as the foundation for a derivative that is used as collateral on a loan and subject to an insurance contract. All of these services would be functionally interoperable, and the resulting instrument benefits from the common settlement layer of the underlying blockchain – but also faces common vulnerabilities.¹⁷ As the number of DeFi service providers and available protocols grows and competition increases, specialization, interoperability and composability can enable growth in the connection between these services, and the economic activity between them.

1.2 The DeFi architecture

Figure 2 is a conceptual overview of the DeFi “stack”.¹⁸ The base-layer blockchain system enables participants to securely store, exchange and modify asset ownership information, replacing the *execution and settlement* layer of conventional financial services. It also allows for the creation of digital assets in various forms, which are then incorporated into DeFi *applications*. Additional layers of applications may function as *aggregators*,

allowing users to shift among DeFi services, such as choosing an exchange based on real-time market factors. In this environment, digital assets may be transferred freely, based on contractual logic (financial flows) or they may be restricted from other uses to provide liquidity or collateral (lock-ups). There are also non-financial information flows that support the transaction activity.

FIGURE 2 | The DeFi stack



Information and content external to the blockchain may also be incorporated into DeFi transaction flows through *oracle* services, which supply reliable data that is recorded outside the settlement layer. For example, a price feed may draw on external data and be delivered programmatically through a

smart contract. Such informational resources, as well as the wallet software and interfaces that help users store, transfer and manage assets interacting with DeFi services, are not themselves financial services and therefore we label them as *auxiliary* to DeFi.

Decentralized governance

Another dimension of the DeFi environment, not shown in Figure 2, is the implementation of decentralized governance mechanisms. Governance refers to the ways in which collective decisions are made, conflicts are resolved and changes to protocols are implemented. In DeFi, governance mediates activity between the applications and underlying settlement layer, including decisions such as altering interest rates or collateral requirements.

This new model raises several new questions for policy-makers and regulators, including:

- How are decisions made?
- How does accountability work?
- How does performance management work?

Many DeFi projects include a *governance token* that provides voting rights on certain governance decisions. Often these tokens are tradeable on exchanges, their value tied to scarcity and the activity level of the issuing DeFi service. Regulators will need to determine the appropriate classification of such tokens. It will be important to evaluate whether tokens are actually employed for governance or simply as a proxy for investment in the service.

FIGURE 3 | DeFi governance approaches

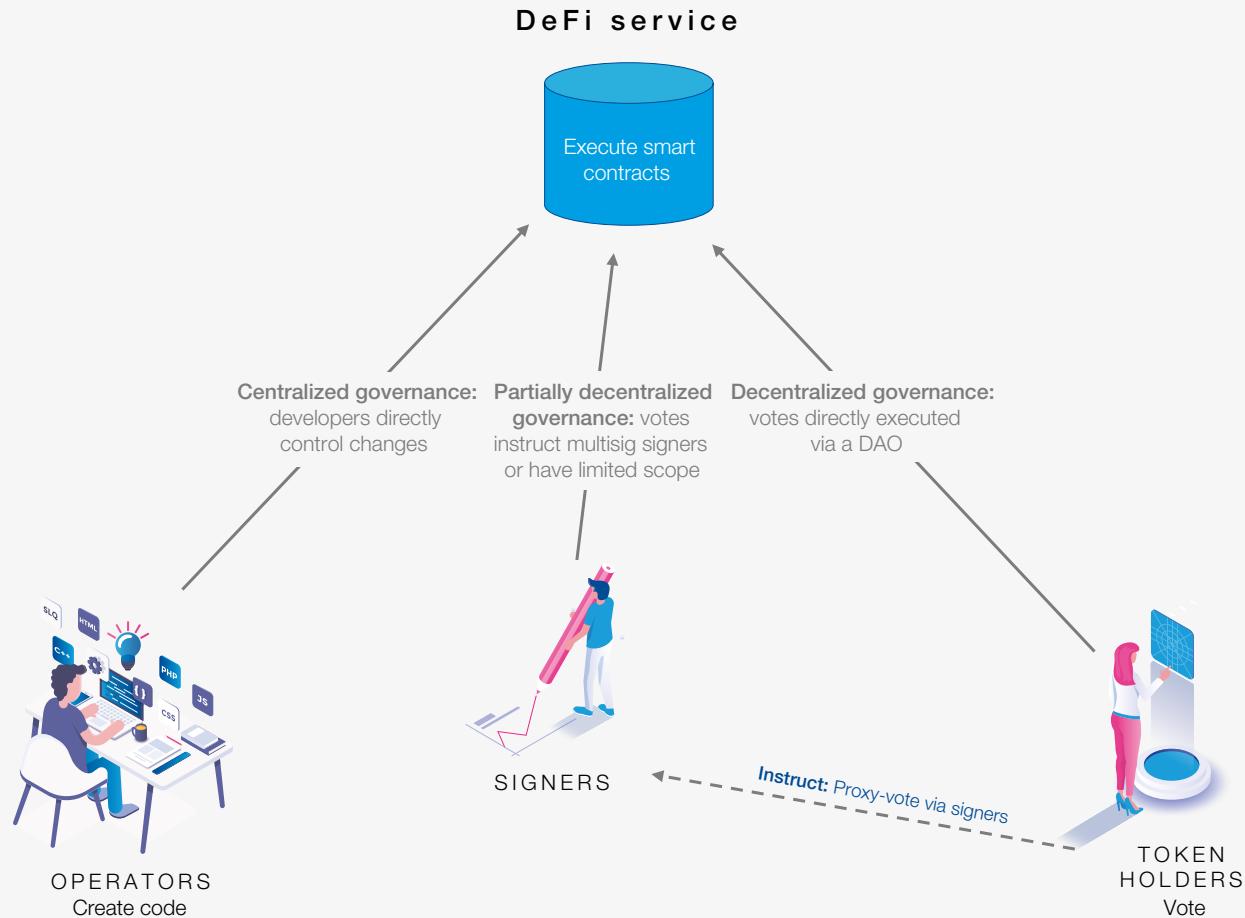


Figure 3 illustrates three forms of DeFi governance. The initial implementation is typically *centralized governance*, where the operator controls and implements changes directly.

Governance can be *partially decentralized* by giving token holders limited voting rights. They may have power over only a few parameters; developers may retain effective veto power through large token holdings or developers may have no formal obligation to implement proposed changes. In some instances of partial decentralization, individuals are designated to implement changes based on the instruction of token holder votes. They do so through *multisig keys*, wherein multiple signatures of delegates are needed to implement a change.

In *decentralized governance*, decisions move fully to a community of token holders through the establishment of a decentralized autonomous organization (DAO). DAO participants vote on

changes to the protocol and are aligned through token incentives and rules written into smart contracts. Governance decisions are executed as blockchain transactions, enforced through the consensus mechanisms of the settlement layer.

DeFi developers often describe a trajectory from centralized governance at the outset to partially and then fully decentralized governance as the service reaches maturity. At this early stage of the market, however, there are few if any examples of this process unfolding from start to finish. The token-based voting systems that have been implemented are immature, and governance votes of major services have failed due to insufficient turnout.¹⁹

Token-based mechanisms for liquidity and governance expand the scope of interested parties beyond those in traditional financial services. Policy-makers should consider the implications of decisions on all of these stakeholder groups, and

the incentives they create – especially considering: (1) who has control of the assets; and (2) who stands to benefit financially. **Appendix 2** provides a stakeholder mapping tool for DeFi services.

BOX 1

DeFi incentive systems

Many DeFi services incorporate explicit financial incentives to promote market development, including the creation of liquidity (for trading) and collateral (for credit):

- **Lock-up yields** pay interest or a share of trading fees for immobilizing digital assets to serve as liquidity or collateral for a service.
- **Liquidity mining** pays interest in the form of tokens issued by the service itself, typically governance tokens.
- **Airdrops** reward wallet addresses with tokens to promote awareness of new digital assets.

- **Yield farming** optimizes returns from liquidity mining and lock-up yields by automatically moving funds among services.
- **Liquidation fees** pay market-makers a percentage of the value of under-collateralized loans that they successfully liquidate (though not necessarily automatically).

These mechanisms are not necessary components of DeFi but have become widely identified with it. However, they may also distort investor expectations, generating unsustainably high returns as new capital is flowing in and token values are appreciating.

1.3 DeFi service categories

Due to their programmability and composability, the possible configurations of DeFi services are nearly endless. However, certain core functions, analogous to those in centralized finance, can be identified. These labels are generic and not intended as regulatory classifications for jurisdictions in which the terms used have legal import. A companion report, [DeFi Beyond the Hype](#), provides greater detail on each of these categories.

Stablecoins seek to maintain a constant value for tokens relative to some stable asset – most commonly the US dollar. The ability to avoid the volatility of non-stabilized cryptocurrency such as bitcoin and ether is one reason for the growth in DeFi.

Custodial stablecoins use holdings of fiat currency or high-quality liquid assets as a reserve. Though they may be used in DeFi, these stablecoins are not DeFi services themselves because they involve centralized trust and custody.

There are two forms of stablecoin that meet the DeFi requirements listed in **Figure 1**:

- *Asset-backed* stablecoins use smart contracts to aggregate and liquidate collateral in the form of digital assets.
- *Algorithmic* stablecoins attempt to maintain the peg through dynamic expansion and contraction of token supply.²⁰

Exchanges allow customers to trade one digital asset for another. The assets involved may be stablecoins or floating-value tokens. Unlike centralized exchanges such as Coinbase or Binance, *decentralized exchange* (DEX) protocols are DeFi services because they do not take custody of user funds and may not control other aspects of the process such as order book management and matching. An important category of DEX protocols for DeFi are *automated market-makers* (AMM), where an algorithm continuously prices transactions based on orders and available liquidity, rather than matching through an order book.

Credit²¹ involves the creation of interest-bearing instruments that must be repaid at maturity. It is based on a mutual relationship of borrowers and lenders, which can be either bilateral (peer-to-peer) or based on pooled capital. Credit terms can be quite complex, and these instruments can themselves be securitized and traded. DeFi borrowing and lending replaces the intermediating function of financial service providers with automated, decentralized, non-custodial protocols. While the lack of credit ratings and legal recourse means that digital asset loans are nearly always over-collateralized, DeFi also allows for uncollateralized *flash loans* in which assets are borrowed and repaid (with interest) within the span of a single block's time.

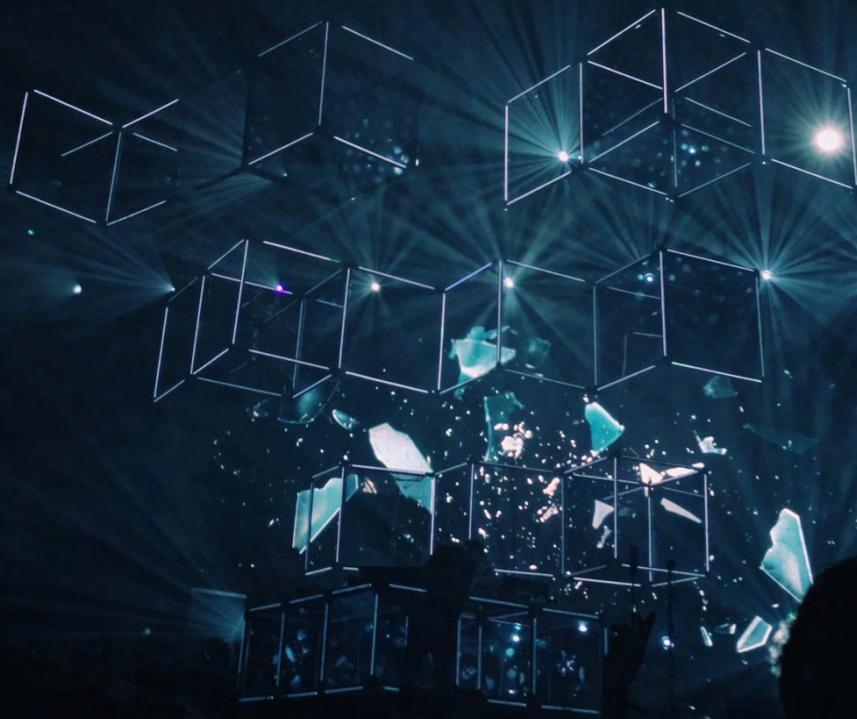
Derivatives create synthetic financial assets whose value is reliant upon or derived from an underlying asset or group of assets. Common financial derivatives include futures and options, which pay out based on the value of an asset at some time in the future or deliver the underlying asset. DeFi derivatives can be programmed and composed into virtually any configuration. For example, a derivative could create a synthetic asset that behaves as a stock, commodity, swap or another digital asset. It could involve a non-fungible token (NFT) uniquely associated with an art or real estate asset. It might be tied to the activity of a business, creating a *crowdfunding* service. Or the value could be tied to a future real-world event, such as the outcome of a sporting event or political campaign, turning the derivatives exchange into a *prediction market*. Prediction markets may also incentivize decentralized information generation or dispute resolution through the wisdom of crowds.

Insurance pools risk by trading the payment of a guaranteed small premium for the possibility of collecting a large payout in the event of a covered scenario. In DeFi insurance, decentralized

transactional and governance systems are used to manage and structure the insurance life cycle for certain types of risks such as smart contract hacks. Though technically insurance contracts are derivatives – they pay out based on some external event – insurance plays a distinctive risk-hedging function in markets by spreading risks across a common capital pool.²²

Asset management involves the oversight of financial assets for others and seeks to maximize the value of the whole portfolio based on risk preferences, time horizons or other conditions. DeFi asset management promises greater transparency and efficiency in constructing and executing investment strategies, by incorporating the asset management life cycle into a DApp.

In addition, there are **auxiliary services** that support DeFi activity but are not themselves financial services. The most prominent are *oracles* (outlined above). Other auxiliary services include wallets, data storage, data queries, identity verification and arbitration.





This section provides a risk-mapping framework as a basis for policy considerations. It contains two stages: (1) identification of relevant risks; and (2) assessment of how DeFi market participants and others are addressing such risks.

We categorize DeFi risks into five categories (explored in more detail below):

Category	Associated risks
Financial Depletion of funds due to the transactional behaviour of fellow users concerning the digital assets in the DeFi service	Market risk Counterparty risk Liquidity risk
Technical Failures of the software systems supporting transaction execution, pricing and integrity	Transaction risk Smart contract risk Miner risk Oracle risk
Operational Failures of the human systems for key management, protocol development or governance	Routine maintenance and upgrades Forks Key management Governance mechanisms Redress of disputes
Legal compliance Use of DeFi to engage in illicit activity or to evade regulatory obligations	Financial crime Fraud and market manipulation Regulatory arbitrage
Emergent Macro-scale crashes or undermining of the financial system due to the interaction, scaling and integration of DeFi components	Dynamic interactions Flash crashes or price cascades
These categories are not mutually exclusive; some failures may result from multiple risks. There are also concerns inherent in the use of blockchains for settlement. For example, proof-of-work blockchains such as Bitcoin and Ethereum version 1.0 require computationally intensive mining, which raises concerns about energy usage that contributes to climate change. Because these issues are not unique to DeFi, they are beyond the scope of this report. ²³	Funds may be lost either unintentionally or due to deliberate attacks. Smart contracts do not distinguish intent and even undesired transactions may be effectively impossible to reverse. This problem was already evident in the 2016 draining of funds from the DAO, ²⁴ the first DeFi service to accrue significant capital. ²⁵ Finally, in some cases, the line between a legitimate trading strategy that takes advantage of an arbitrage opportunity and an improper exploit might be unclear.

2.1 | Financial

Market risk is the possibility that asset value will decline over some time horizon due to market conditions, new information or traders' idiosyncratic behaviour. Though it may not be the role of governments to protect against market risk for well-informed and well-capitalized investors in a well-functioning market, it is appropriate for them to be concerned that those conditions are met. For DeFi, regulatory classifications will define whether requirements designed to prevent undue market risk – such as disclosure obligations and accredited investor standards – are applicable.

DeFi's novelty, as well as the ease of transferring funds and creating complex instruments, may increase the possibility of abuses, whether by the creators of DeFi protocols, the operators of exchanges or third-party manipulators. At the same time, policy-makers may want to consider the implications of potential increases in transparency as well as the retention of asset custody. There may also be a lack of observability and standardized price-discovery mechanisms found in digital asset markets. The inability to compare many of the current tokens to any fundamentals is a driver of big swings in valuation and overall volatility.

Counterparty risk is the possibility that a counterparty will default on its obligations to a financial instrument. This might involve failing to repay a loan (*credit risk*) or failing to settle a transaction by providing the specified asset (*settlement risk*). Though some credit risk is mitigated through interest rates for loans, it might be a particular problem in DeFi, where the volatility of underlying digital assets produces under-collateralization, the ease of credit creation leads to excessive leverage, or the algorithmic determination of interest produces inaccuracies. The lack of fixed identities in a pseudonymous network presents additional challenges in terms of determining creditworthiness. DeFi attempts to account for this through over-collateralization requirements.

Some traditional settlement risks are not present in DeFi because there is no separate settlement step; transactions are executed through transfer of the underlying value on the blockchain – but only if both sides of the transaction are operating on the same chain. Moreover, given the rapid inflow of capital, there are strong incentives and many opportunities for scams. Users may not receive the assets they anticipate due to fraud, especially when information asymmetries limit their understanding of investment decisions or the code that governs transaction execution.

Liquidity risk is the possibility that there will be insufficient funds or assets available to realize the value of a financial asset. Failure of liquidity for a borrower or trader (such as a short seller) means the position is involuntarily liquidated and the available assets allocated to owners or creditors. Insufficient liquidity also magnifies market inefficiencies, such as price movements resulting from trades.

DeFi liquidation processes differ from traditional instruments, where a centralized counterparty (a bank, the International Swaps and Derivatives Association, a clearing house, etc.) executes the process. DeFi services often incentivize market-makers to liquidate under-collateralized loans, performing a function analogous to a foreclosure auction for real estate. If the liquidation incentive structures fail, however, original counterparties and liquidity providers hold unanticipated default risk. In DeFi markets where most transactions are automated and available continuously, the speed of liquidations may preclude rational decision-making. On centralized exchanges, cascades of automated liquidations have on several occasions produced "flash crashes", where prices dropped precipitously and trading was taken offline until the market settled. Such last-resort remedies may not be available for decentralized services.

DeFi liquidity risks may be mitigated through governance logic and the careful design of incentive structures. Game-theoretic analysis must anticipate not only expected behaviours, but other profitable strategies. For example, a market participant could deliberately skew liquidity in certain DeFi services and bet against the arbitrary results. Systems designed to incentivize stable liquidity could limit this risk. Because financial risks arise from profit-seeking, constant vigilance is needed to address new strategies.

Flash loans create a unique set of risks. They may effectively create artificial liquidity for a short period of time, seemingly addressing both counterparty and liquidity risk. If the loan cannot be paid back in time, the original transaction is never incorporated into the block and the loan is essentially rolled back before issuance. While flash loans may be used as near risk-free and low-cost capital for legitimate arbitrage transactions, they can also be employed in attacks. The temporary surge of funds can be used to manipulate prices and force artificial liquidation, often through the interaction of multiple DeFi services. Several million dollars have been stolen through several such high-profile, near-instantaneous attacks.²⁶

2.2 | Technical

According to Ciphertrace, half of digital asset hacks in 2020 targeted DeFi services, up from a negligible number in 2019 – a trend likely to continue as the value of assets involved grows.²⁷ While the largest public blockchain networks, such as Bitcoin and Ethereum, have avoided significant breaches, blockchain-based DApps and the centralized exchanges or wallets handling funds have proven far less secure. The technical complexity and immaturity of the DeFi market increases the likelihood of significant vulnerabilities, with the vast majority created in the past few years. The degree of interconnection among DeFi protocols may also expand the attack surface available to malicious actors.

Services aim to police market abuses through radical transparency and trust minimization rather than centralized oversight. Some include sophisticated, multilayered incentive structures to discourage attacks, in addition to technical measures for security and market integrity. Some have used their decentralized governance mechanisms to implement changes in response to failures or potential scenarios identified by the community. These measures are not foolproof. If DeFi continues to grow and attract less sophisticated market participants, investor protection concerns may grow.

Transaction risks are limitations or failures of the underlying blockchain network. If the base-layer settlement network is successfully attacked, allows for double-spending, becomes too expensive for

transactions or lacks the necessary throughput, those failures will affect the application layer. The long-planned upgrade to Eth2 (Ethereum version 2.0), which aims for significant performance improvements, thus represents an important development for DeFi.²⁸ This upgrade will also shift Ethereum to proof-of-stake consensus, which does not require the intensive energy usage of proof-of-work mining.

Smart contract risks deal with code that does not execute as intended. All software has the potential for bugs. A programming flaw can cause a smart contract to fail to perform as desired, or attackers can exploit vulnerabilities to drain funds or engage in malicious activities. For example, where code has not been written properly, it can allow for exploits such as re-entrancy attacks. Complex software performing novel functions in a relatively untested environment, and often written by teams lacking the expertise or inclination to employ the most robust development practices, will tend to have more bugs than the norm.²⁹ Even without attacks, the smart contract might not accurately reflect the understanding of all parties. Because DeFi software is automated financial services, rather than a record-keeping mechanism subject to human override, coding errors can lead directly to financial losses, often without easy redress. Moreover, transparency of code has two sides – the visibility may make smart contracts more vulnerable to exploits or may offer opportunities for white hat hackers and bounty hunters to increase the robustness of the code.

CASE STUDY

The DAO exploit

The DAO, a decentralized crowdfunding platform, was arguably the first viable DeFi service. In 2016, ether then worth approximately \$150 million was locked up in its smart contracts, with the goal of funding decentralized application development.³⁰ Before it launched, however, an attacker exploited a re-entrancy bug to drain approximately 40%

of the funds into a “child DAO”. To prevent permanent loss, and the collapse of confidence in Ethereum, miners agreed to implement a hard fork that reversed the theft on the main Ethereum chain. A minority faction continued mining the deprecated chain, which became known as Ethereum Classic.

Mechanisms such as security audits and bug bounties can be employed to mitigate smart contract risks. Over time, common errors in smart contracts written in popular languages such as Ethereum’s Solidity become more familiar, and high-quality teams know to look for common attack vectors.

Miner risk deals with the possibility that transaction processing entities behave maliciously towards certain transactions. This depends on the correct ordering and execution of transactions sent to a DeFi smart contract. It operates at an analogous level to settlement risk in centralized finance, involving the finalization of transactions, although the nature of the threat is different. In blockchain systems, users typically send a transaction to the network along with a fee to the miner that successfully processes it into a block.

Miners take proposed transactions and decide the order in which to execute them. However, a miner need not execute transactions in fee order. A miner can choose to execute a lower-fee transaction ahead of a higher-fee transaction, if that transaction is particularly valuable to them, or in return for a side payment from the originator of the lower-fee transaction.

Such behaviour allows for a form of market manipulation like front-running in high-frequency trading. By manipulating the order of execution, a miner can effectively allow certain parties to compound returns faster than others. Some view “miner extractable value” as inevitable in any system based on public blockchains, which is legitimate if structured transparently and fairly. This is a topic of active debate in the DeFi community.³¹

DEX arbitrage bots

Researchers have documented and quantified the rising deployment of arbitrage bots in decentralized exchanges.³² Like high-frequency traders on Wall Street, these bots exploit inefficiencies, paying high transaction fees and optimizing network latency to front-run (anticipate and exploit) ordinary users' DEX trades. They

study the breadth of DEX arbitrage bots in a subset of transactions that yield quantifiable revenue to these bots by engaging in priority gas auctions (PGAs), competitively bidding up transaction fees in order to obtain priority ordering, i.e. early block position and execution, for their transactions.

Oracle risk involves the potential that data external to the blockchain on which a DeFi contract relies is inaccurate or has been manipulated. Oracle-dependent DeFi protocols are susceptible to attacks in which oracle providers can manipulate the price observed on-chain. If on-chain asset holders can do this, they can increase the value of their on-chain asset or decrease the value of other participants' assets. Re-marking below a liquidation threshold could lead to assets being sold to the highest or first bidder.

If an oracle uses a centralized data source, such as a feed from CoinMarketCap for prices, this represents a source of centralized trust and vulnerability. An oracle can be decentralized by using multiple data sources or by incentivizing providers to submit data. Decentralization makes it difficult for a small number of participants to manipulate prices. On the other hand, payments to data providers must be designed effectively for fairness and incentive compatibility to ensure accurate information. Poor mechanism design may make it profitable to manipulate oracle data feeds. There have already been several successful DeFi oracle attacks.

The Compound oracle exploit

In November 2020, the price of the DAI stablecoin was temporarily driven up 30% over its \$1 peg on the Coinbase exchange, which was used as the pricing oracle by the Compound DeFi credit platform.³³ When the DAI price spiked, it caused Compound's smart contracts to determine that many loans were under-collateralized. This triggered \$89 million of assets locked in Compound to be liquidated automatically. It is

unclear what caused the anomalous increase in the Coinbase price, but it could have been an intentional form of manipulation directed at Compound. This event illustrated the risks inherent in the interconnection among DeFi and other blockchain-based financial systems – and that some elements of the ecosystem may not be as decentralized, and therefore more vulnerable, than it initially appears.

2.3 Operational

Even though DeFi activity is highly automated, human operators still play a crucial role. The more decentralized a service, the less risk there is associated with any single point of failure. Auxiliary services may be centralized even when the DeFi service is highly decentralized. At the same time, greater decentralization can make it harder to respond effectively when something goes wrong. The fewer people who have unique power to break a service, the fewer who have the power to fix it.

Routine maintenance and upgrades may be more difficult to implement for decentralized services, or may create vulnerabilities, especially

given the composability of DeFi. This would also include ongoing network and node connectivity and considerations related to security and cyber risks.

Code forks are options for groups seeking to alter elements of DeFi services, providing an “exit” option for minorities that prefer a different set of parameters.³⁴ In some cases, a fork may become more popular than the original service. When there is already significant activity on a platform, however, forks can be costly and confusing for participants. They can also be employed for malicious purposes, including to mislead users.

SushiSwap vampire attack on Uniswap

In September 2020, a pseudonymous developer, Chef Nomi, forked Uniswap, an open-source decentralized exchange, to make SushiSwap, a nearly identical exchange with an added token (SUSHI) and token rewards for liquidity providers and token holders.³⁵ The incident, which became known as the first “vampire mining” event, was unique in that SushiSwap indirectly competed with Uniswap by providing the same service using identical code but with an additional incentive,

draining Uniswap’s liquidity. Initial participants in SushiSwap earned SUSHI by depositing Uniswap’s LP tokens, which represented user deposits in the Uniswap DEX. These Uniswap LP tokens were then swapped for the SUSHI, so that Uniswap liquidity would become SushiSwap liquidity. Ten days later, the pseudonymous developer sold all of his SUSHI tokens for \$13 million in Ether and handed over control of the protocol to the Chief Executive Officer of FTX, a centralized exchange.

Key management is a potential problem for all blockchain-based systems. Platforms identify users and their assets through cryptographic key pairs. Because DeFi services are non-custodial, they place the key management burden on their users in return for removing dependencies on centralized service providers. A variety of techniques including requiring multiple signatures (multisig), social recovery and custody arrangements have been developed to address key management risks for digital assets.

Governance mechanisms for DeFi and other blockchain-based services raise complex potential risks. “One-token, one-vote” may be exploited when participation rates are low, token control is concentrated or participants can bribe each other

to vote in their favour. Centralized exchanges may take advantage of the voting power of tokens in their custody to exert undue influence in governance. Specialized DeFi market participants may engage in activities analogous to activist investing, deliberately acquiring significant shares of governance tokens for a service. With enough voting power, these investors could change the parameters, allowing them to drain liquidity pools. Even though many of the mechanisms incorporated into DeFi governance systems have a history in academic literature, their behaviour with large numbers of participants and millions or billions of dollars at stake remains unproven. Moreover, a recent research paper presents evidence that DeFi token holdings are heavily concentrated, in ways that are not entirely transparent.³⁶

Flash loans and MakerDAO governance

Flash loans also pose challenges for governance systems. In late October 2020, an attacker used a flash loan to acquire \$7 million of the MKR governance token associated with the MakerDAO protocol and exercised its rights to

vote on a governance proposal. Concerned about the potential for abuse, MakerDAO adopted restrictions shortly thereafter to prevent this scenario from being repeated, but other DeFi services remain vulnerable to such attacks.

Redress of disputes is a final category of governance risks. Once a smart contract has executed, the output cannot be modified or reversed just because an individual actor, or a governmental authority, orders it to be. When participants believe they are entitled to redress for some failure of the system or malicious act, arbitration may be incorporated into the DeFi service through multisig arrangements or be decentralized through a prediction market or

crowdsourcing mechanism. However, these novel mechanisms have their own limitations, for instance, compared to judicial or administrative orders.

With a well-designed DeFi service, operational risks may be measured in real time and actively mitigated. DeFi transaction ledgers are public, so malicious activities may be tracked more easily than in analogous cases for centralized finance.



2.4 Legal compliance

DeFi may be used to bypass legal or regulatory obligations. The activities involved could occur with any service involving digital assets. Money laundering, for example, is a problem for established centralized cryptocurrency exchanges as well as DeFi DEXs. Because the focus of this report is on the distinctive challenges and opportunities of DeFi, we provide only a brief summary of risks in this category.

While a DeFi structure may not increase the likelihood of such violations *per se*, it could complicate enforcement. The decentralized, non-custodial, composable nature of DeFi services may make it difficult to identify a responsible party, for example. Regulatory regimes built around intermediaries as regulated processors of transaction information may fit poorly with a disintermediated market structure. We consider how regulators and policy-makers might address such challenges in Section 3, below.

Financial crime involves breach of anti-money laundering/countering the financing of terrorism (AML/CFT) restrictions, financial sanctions and similar legal regimes. DeFi transactions involving natively digital assets may be difficult to regulate through traditional AML/CFT controls because users are pseudonymous by default, transactions are resistant to blockage, assets are resistant to seizure and many transactions involve non-custodial wallets not directly tied to individuals. Although DeFi transactions are generally transparent and traceable, new privacy-enhancing protocols and/or tools may create additional regulatory challenges. Several approaches have been developed to comply with the 2019 anti-money

laundering guidance for digital asset service providers from the Financial Action Task Force (FATF), but further work remains and could be further affected by new guidance proposed in March 2021 that could require know-your-customer (KYC) compliance from DeFi services.³⁷ In particular, the use of non-custodial arrangements and self-hosted wallets in DeFi poses a challenge for requirements that identifying metadata be collected and passed for every transaction link.

Fraud and market manipulation involve deliberate scams, misappropriation and other efforts to take advantage of investors. Here we refer to activities conducted or enabled by DeFi developers themselves, rather than third-party attacks. For example, “rug pulls” or exit scams involve convincing users to place funds into a seemingly legitimate DeFi service, from which they are drained by the developers, who then disappear.

Regulatory evasion means failing to meet regulatory obligations by carrying out similar functions in a different technical manner. It may involve deliberately obfuscating activity or masking the jurisdictional attributes of transactions. On the other hand, the fact that a novel activity bears similarities to an established one does not automatically imply regulatory arbitrage. Poorly designed regulatory obligations could themselves be viewed as a risk factor for DeFi. All major categories of DeFi activity can be viewed as alternatives to regulated financial services. Whether they are subject to similar classifications goes beyond the scope of this report, and the answers will vary by jurisdiction.

2.5 Emergent risks

Emergent risks involve the interaction effects of multiple events, creating failure cases that are not reflected in a risk assessment of each service independently. Classic recent examples are banks that are “too big to fail” and scenarios in which ostensibly unrelated events, such as individual mortgage defaults, become highly correlated and produce cascading effects through chains of securitization. Other examples include system-wide liquidity failure due to bank runs or markets “freezing up” when parties are unwilling to transact due to perceived risk.

Dynamic interactions among a potentially endless number of interconnected DeFi components may produce risks that are not present in any individual service. Also, because DeFi operates in a global market, activities are not necessarily limited to countries or business segments as they are when transactions are based on a national sovereign currency. Unless regulators can effectively limit

cross-border DeFi activity, firebreaks to contagion of systemic defaults may be more limited than for traditional finance. Interaction risks will also grow as DeFi services begin to interoperate with traditional financial platforms.³⁸

Flash crashes or price cascades, exacerbated by leverage in the DeFi system, may occur in extremely volatile or rough market conditions. Unlike traditional markets, where primary dealers and brokers can manually intervene when defaults occur concurrently, the permissionless, algorithmic nature of DeFi means that it may not be possible to stop cascades. DeFi services that automatically liquidate collateral allow liquidators to compete to buy that collateral, sometimes offering a fixed discount as an incentive. However, when a flash crash occurs or market volatility is high, there may be so many liquidations and the drop in the price of the collateral may be so precipitous that liquidators or others will face significant losses.

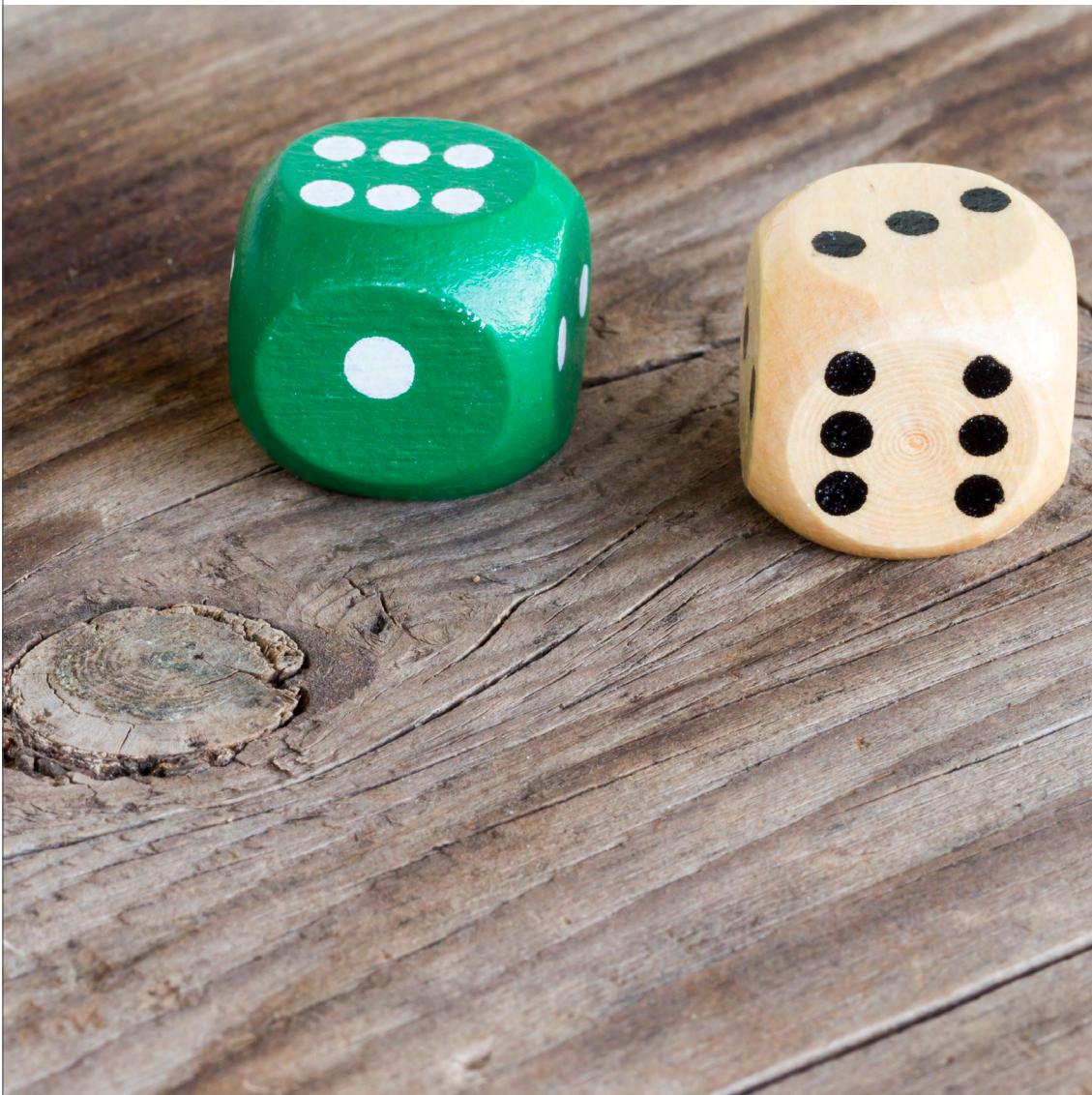
One of the largest systemic failures of a DeFi service took place on Thursday 12 March 2020, when Maker Protocol's liquidation system failed and more than \$8 million in user assets was lost.³⁹ This was exacerbated by network congestion on the Ethereum blockchain, which increased "gas" prices for validating transactions and slowed the flow of data updates to MakerDAO's oracle service.

A class-action lawsuit over the event, claiming that the Maker Foundation deceived collateral providers by failing to appropriately disclose such risks, was sent to arbitration in September 2020. The event exposed emergent risks faced by the DeFi protocols, including the availability and reliability of the underlying blockchain infrastructure.

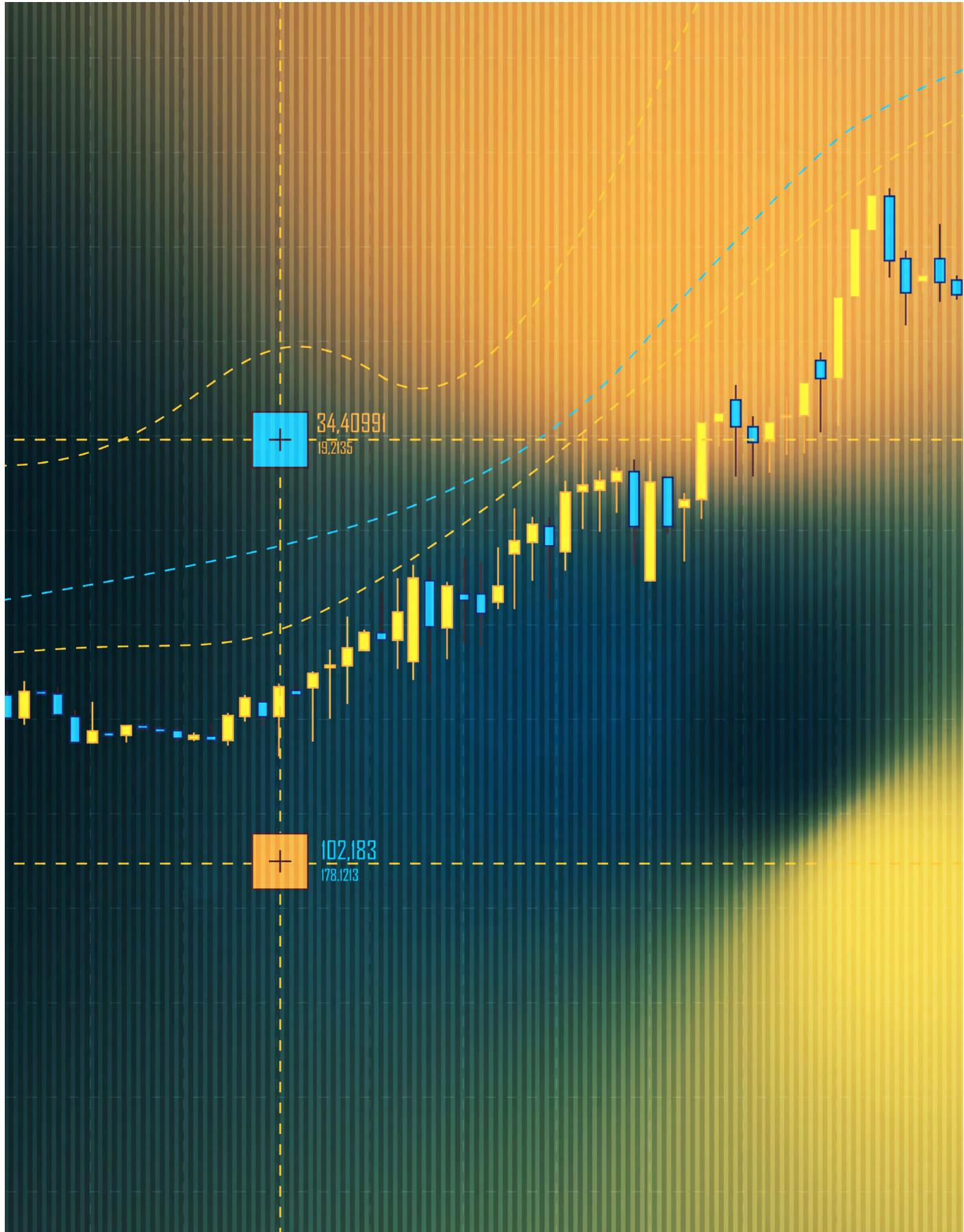
Assessing such risk is difficult. Most traditional financial models assume that liquidations always occur successfully, as the trusted third party (exchange, broker, dealer) will close a position when unprofitable. In DeFi, this is true only when liquidators can achieve a profitable liquidation. If cascades persist for too long, liquidators stop liquidating and traditional value-at-risk (VaR) models break down. This failure is akin to what happened during the 2008 financial crisis, when centralized third parties that enforced liquidations, such as AIG, failed.

Risks of this form can be estimated using tools such as agent-based simulation, which model

rational behaviour for all principal parties (borrowers, lenders, traders and liquidators) and then run millions of event-based Monte Carlo simulations – models for predicting outcomes for situations subject to random variables – to estimate worst-case loss. Unlike traditional financial Monte Carlo simulations, these simulations explore conditions in which financial assumptions such as no-arbitrage and instant liquidation are invalid. Using such models, corrections to traditional value-at-risk (VAR) models can be estimated, leading to estimates of default probability as a function of parameters such as volatility.



Policy approaches



This section outlines the main areas in which DeFi may interact with policy and regulation. Importantly, it lays out key issues and options but does not offer prescriptive solutions, as jurisdictions vary in their objectives, regulatory regimes and market composition. The approaches described here are intended to be sufficiently generic to apply in the full range of contexts. The remainder of this section provides tools and resources.

Trust-minimized execution, non-custodial services and composable architectures may challenge the existing regulation. As described in Section 2, DeFi can both introduce new risks and may help mitigate known risks in financial services. Many of the key challenges for policy-makers – the way in which decentralization makes it difficult to identify regulatory subjects, new risks due to automation, and the way in which borderless software code

complicates the application of territorial rules – are extensions of issues for all digital assets. Others, such as the creation of building blocks with multiple potential use cases and integrations, or the incentive structures of tokenized governance, are less familiar. Given the cross-cutting nature of DeFi, an integrated strategy and vision is needed.

Generally, it may be wise to consider a **technologically neutral** approach to balance meeting the objectives of regulatory regimes with promoting innovation and market development. As with any regulatory initiatives, policy-makers should strive for DeFi rules that are fair, efficient, effective and enforceable.

A policy-maker canvas, included as **Appendix 4**, is designed to apply key components of this section in a structured manner.

3.1 DeFi and financial regulation

The first step is to identify the relevant objectives and associated categories of policy and regulation. Common goals for financial regulation include: protection of investors and other consumers; market efficiency and integrity; capital formation; financial inclusion; prevention of illicit activity; safety and soundness; and financial stability. Each provides a distinctive logic for certain kinds of rules. For example, regulators focused on investor protection are typically concerned that custodians are not able to abscond with funds. The non-custodial nature of DeFi may alleviate some of these worries, while creating new ones (as outlined in Section 2).

DeFi activity spans many domains of financial regulation, including securities, derivatives, exchanges, investment management, bank supervision, financial crime, consumer finance, insurance, risk management and macroprudential oversight. A coherent overarching strategy is important and could be delegated to a cross-entity taskforce or similar body. Some DeFi activity patterns will clearly match established legal categories; others will not.

A range of policy actions may be adopted for DeFi, including:

- **Forbearance:** decision that no new regulations are needed
- **Warnings:** issuance of warning to users/consumers
- **Enforcement:** determinations that existing rules already cover the relevant actors and activities and have not been complied with

- **Opt-in:** provide the option to become subject to regulations in return for certain protections, even though there is no legal requirement
- **Pruning regulations:** eliminate regulatory requirements that are no longer essential in a DeFi context
- **Limited licence frameworks:** the possibility of obtaining licences of limited scope or under size thresholds, with light-touch requirements
- **Prohibitive measures:** prohibit certain activities in the DeFi sector
- **New licence types:** address risks with new categories designed for DeFi
- **Issuing guidance or expectations:** craft new frameworks, often with a public comment or consultation included before its official release

An effective regulatory response to DeFi is likely to involve a combination of existing regulation, retrofitted regulation and new, bespoke regulation.⁴⁰ An emerging body of digital asset-specific law is growing, including the European Union's comprehensive Markets in Crypto Assets (MiCA) proposal.⁴¹ However, most jurisdictions are yet to adopt bespoke frameworks.

Most financial regulatory regimes focus on those “carrying on business” in a certain regulated activity, “dealing”, “arranging” or “operating” some scheme or exchange or “issuing” an offer (or similar). Historically, the relevant government entity was relatively clear and focused on who is ultimately in control of an operation. Similarly, there are often

exemptions for service providers that merely provide infrastructure, data or other tools to enable others to layer on their financial services. Frameworks contemplate definable and centralized operators that are engaged in providing particular financial end products and services, but are not necessarily the underlying builders.

In the DeFi context, however, there may be no central entity performing the relevant activities. The software developers and token holders may be easily identifiable, but not those occupying roles that are the traditional regulatory touchpoints. And even when operators can be identified, they may lack the ability to modify DeFi services or stop transactions because of the decentralized nature of the protocols. Smart contracts can interact with assets held by other smart contracts that are not directly associated with a particular user. Regulators will need to assess who is *responsible* and when a locus of responsibility must be identified. It may

be possible to do so through careful analysis of services, even when they are nominally decentralized.⁴²

Legal regimes often include mechanisms for vicarious secondary “controlling person”, “responsible officer” or aiding-and-abetting liability based on requirements such as knowledge or foresight of harmful consequences.⁴³ If developers of a DeFi service or others associated with the DeFi business *could have* identified and mitigated legal compliance risks, policy-makers will need to consider whether it is appropriate to mandate that they should have. On the other hand, regulating the creation of software raises important concerns of freedom of speech and administrability, which should be considered carefully. The borderless nature of blockchain networks and digital assets also poses challenges for DeFi regulation at the national or subnational level.

3.2 Available policy tools

There are many ways for a policy-maker to approach new financial services or products. Below, we first identify some helpful steps that regulators

have taken in responding to the rise of digital assets and token offerings.

1. Transitional mechanisms

While not entirely analogous, policy approaches may be informed by how digital assets were initially addressed. In the 2017 initial coin offerings (ICOs) boom, few regulators had structures or expertise fit for purpose as – seemingly out of nowhere – significant capital was flowing into new platforms that claimed to be outside the regulatory perimeter. Some of the initial responses may prove useful in the context of DeFi.

Specialized regulatory units: A targeted desk with qualified staffing can serve as an initial gateway to gain experience in new technology, interact with the industry and provide guidance. This knowledge can be shared with policy-makers and actions may include issuing non-action letters under existing regulatory regimes. These groups may provide legal clarity to DeFi projects and encourage early-stage discussions with regulators. Regulators should also invest in technology and technical expertise to understand these markets more effectively. Many jurisdictions have used this approach. For example, the US Securities and Exchange Commission (SEC) created its FinHub unit (upgraded to a formal stand-alone office in late 2020), while Switzerland’s financial regulator, FINMA, created the FinTech Desk. Though initially small and limited in authority, they quickly became an important point of contact for both internal and external communities.

Incentivizing information flow: Disclosure is one of the most common tools of financial regulation. Even when the applicability of existing disclosure requirements on DeFi platforms is uncertain, efforts to encourage broad and consistent information disclosure may prove fruitful for regulatory analysis. The Monetary Authority of Singapore focused a significant portion of its regulation of ICOs on reviews of white papers.

Regulatory sandboxes: Policy-makers may decide to establish regulatory forbearance programmes such as sandboxes, where companies may test and operate their technology in a limited scope and therefore with limited regulatory risks. The scope of such regulatory “carve-outs” can be defined by activities, financial thresholds, territorial or customer limits and combined with reporting duties to ensure that the regulatory authority gains experience in new technology, interacts with the industry and reacts if new risks arise. However, a lack of transparency from the regulatory authority about the trajectory may inadvertently stifle innovation and there may be business risks involved for start-ups building in sandboxes without explicit safe harbours. The sandbox gives start-ups a chance to address regulatory compliance concerns and gives regulators a better understanding of the risks and benefits of a new space. A DeFi sandbox might go

beyond the prior models by establishing a means of monitoring the trajectory for projects looking to decentralize control over time in order to address some concerns without creating new ones. The UK Financial Conduct Authority (FCA) established a sandbox regime for fintech that included a substantial number of blockchain and digital asset services. However, it has had limited applicability for DeFi because stablecoins are considered to be outside the FCA's scope. Others, such as Colombia's "la Arenera" sandbox, have followed this approach as well. DeFi sandboxes will need to be designed carefully to avoid prematurely signalling approval from the regulator.

A variation of this approach is a *regulation-free zone*, as implemented in Busan, South Korea. Under this model, specific jurisdictions within a country may allow companies to operate under a limited set of regulations (often not fully "regulation-free") in order to allow for innovation and testing of services.

Clarifying easy cases. There will always be some new activities that clearly raise regulatory red flags, some that do not and others that are in grey areas. Sometimes by taking on the easier cases first, especially those where intervention is not warranted, policy-makers and regulators can narrow the zone of uncertainty and incentivize compliance activities. A more formal approach for

distinguishing easy cases is a safe harbour policy that explicitly excludes from regulation services that meet defined criteria. In the ICO case, the US SEC's first official statement was the 2017 investigative report on the DAO.⁴⁴ It clarified that bitcoin was not considered a security, but that a token created for investment purposes would be. Further, because the DAO had already shut down, there was no need for an enforcement action. Though it left many questions unanswered, the report clarified the SEC's approach and its concerns, facilitating further dialogue.

Coordinating government action. In some cases, it may be useful to bring together different government entities for a harmonized response. An example is the modification of the "Volcker Rule" in the US by five federal regulatory agencies (the SEC, the Commodity Futures Trading Commission [CFTC], the Federal Deposit Insurance Corporation [FDIC], the Office of the Comptroller of the Currency [OCC] and the Federal Reserve Board).

This list is not intended to be comprehensive. Nor does it presuppose the direction in which the policy-makers will eventually go. These techniques are equally relevant if DeFi services are ultimately found to be covered by existing requirements, outside the regulatory perimeter or subject to new, bespoke rules.

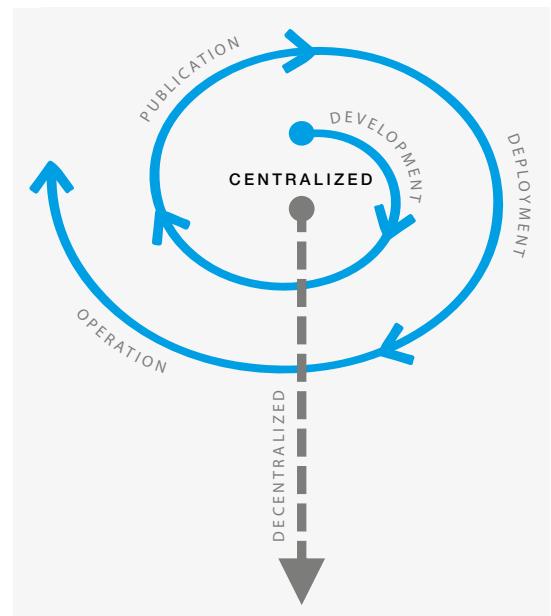
2. Regulation throughout the life cycle

Where there is no inherent distinctive risk, regulation typically occurs later in the life cycle of a product or service, when the harms that trigger liability or regulatory enforcement are more likely to occur. Early on, regulators are more likely to adopt a "do no harm" approach, given the relatively small scale and innovative potential of nascent technologies. For products with clearly known dangers or misuses, such as poppy flowers and weapons, the industry is strictly regulated and all stages are carefully supervised and controlled. Technological systems tend to fall somewhere in the middle.

In addition to maturing, DeFi services have the potential to become more decentralized across their life cycle, as detailed in **Appendix 3**. The degree of decentralization is also an important consideration for policy-makers and regulators. Rules to address the potential dangers of DeFi services can be adopted at four stages of the life cycle: (1) development; (2) publication; (3) deployment; and (4) operation, as shown in **Figure 4**.

There will typically be an identifiable group of protocol developers (although it might operate under the umbrella of an open-source development community, non-profit foundation or association or the DAO). Once the protocol is published, multiple teams might develop it into services and market

FIGURE 4 | DeFi service life cycle



it to users, representing a combined deployment stage. Those services might later be forked by different teams. The operation of the service will largely be automated by the protocol and smart contracts, perhaps moderated by decentralized governance processes.

Imposing regulatory obligations may be easier earlier in the life cycle, where there may be clearly identifiable access points and more room to influence the long-term trajectory. However, the earlier in the life cycle, the weaker the nexus to actual demonstrable harm and the greater the potential

implications for innovation – so it is important to determine at what point regulatory involvement is proportionate to the risks. Tools that incentivize rather than mandate action at early stages, including sandboxes, safe harbours and no-action letters, can be a valuable means of mediating this conflict.

3.3 Decision tree

This tool integrates frameworks presented in this toolkit to support internal policy and regulatory analysis of DeFi services. It is not intended to provide specific recommendations on when and how to act.

This may be used in conjunction with the other resources cited in this toolkit:

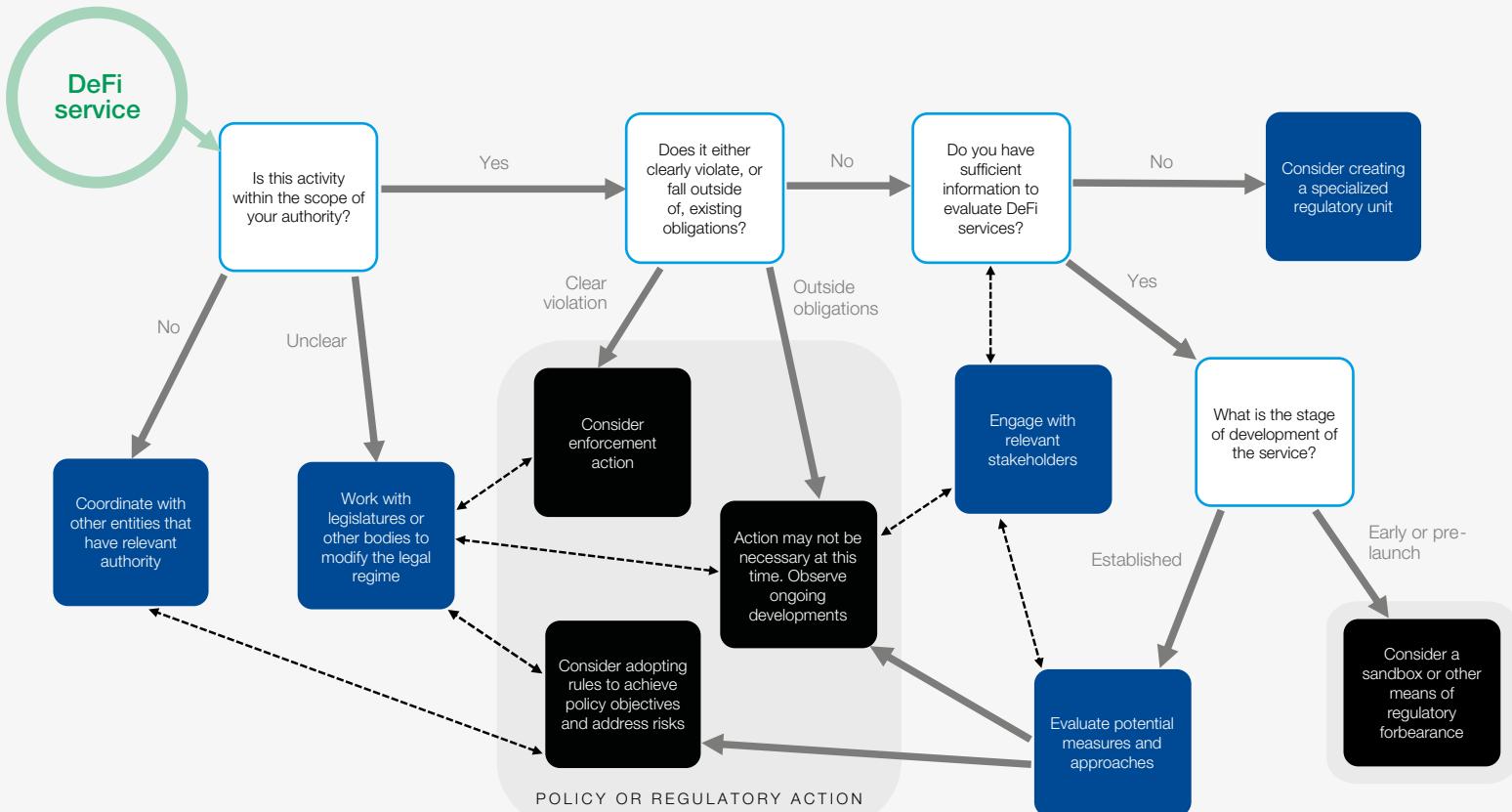
- **Appendix 1** offers a series of questions to identify relevant policy considerations and capabilities for DeFi generally.
- When considering a policy or protocol, the initial step, following **Figure 1**, is to determine whether the activity represents DeFi.
- **Figure 2** and the companion paper *DeFi Beyond the Hype* can be used to understand the relevant service categories and features.

If a service is considered DeFi:

- The questions shown below in **Figure 5** are designed to clarify the suggested courses of action.
- **Appendix 2** is a stakeholder mapping tool that can be used to identify relevant stakeholders for engagement.
- The decentralization spectrum in **Appendix 3** allows for a more precise picture of whether there are significant points of control in the service that might be relevant for decision-making.

Finally, when a determination to consider policy or regulatory action has been made, the policy-maker canvas in **Appendix 4** walks through a series of questions to assist in developing specific responses.

FIGURE 5
Decision tree for evaluating DeFi services



Conclusion

This toolkit is designed as a starting point for policy-makers seeking to understand the risks and opportunities posed by DeFi businesses and services, and to devise the best policy responses. The particular manifestations of DeFi, and the policy questions they pose, will change over time, as will activity levels and other aspects of the larger blockchain and digital asset world.

Policy-makers and regulators will take different approaches based on the unique context of their jurisdictions. Larger shifts in financial regulatory obligations, or implementation of cross-national standards, may alter the context for consideration of DeFi issues. There were no decentralized digital currency assets before 2009, and no general-purpose smart contract platforms before 2015, so

any recommendations about the proper treatment of an offshoot such as DeFi must consider potential and unpredictable developments in a space that is evolving rapidly.

What is clear is that DeFi represents a distinct and potentially significant development, both within the landscape of blockchain and of financial services more generally. As this report has documented, DeFi presents a host of opportunities and many challenges. Even when there are no clear answers, policy-makers are best served by considering the right questions to ask, appreciating the points of interaction and tension with their regulatory regimes, and estimating the costs and benefits of various courses of action.



Appendix 1:

Background assessment

The following questions are designed to help evaluate fundamental background questions before proceeding with policy or regulatory decisions.

Editable versions are available in [Word](#) and [Excel](#) form.

- Is DeFi, or a subset of DeFi services, within your entity's mandate? If so, what are the relevant policy or regulatory areas of focus? What are the top three risks you are focused on?

Top three risks:

1.

2.

3.

- Are there other entities that have relevant mandates? What are they? How do their jurisdictional scope and risk priorities compare to yours? What are your procedures, if any, for coordinating with those entities?
- Has DeFi been explored by your entity or others? What were the outcomes of those explorations?
- What is the in-house knowledge, experience and expertise related to DeFi? What about fundamentals such as digital assets, blockchain technology and decentralized governance?

- What is the process for getting up to speed on quickly evolving spaces and technologies? Are these relevant to DeFi or will they need to be adapted?
- Which parties in the public or private sector are required to provide input or consultation regarding potential changes in policies/regulations related to the financial system and financial technology?
- From which institutions or parties would it be beneficial to solicit input? Which additional stakeholders should be represented and involved in decision-making?

Appendix 2:

Stakeholder mapping tool

This tool is designed to help policy-makers map out the relevant environment of a given DeFi service. We group stakeholders into four categories, though in some cases stakeholders may span multiple categories:

- **Builders:** create, implement and support DeFi protocol
 - **Suppliers:** provide capital or a core service to the functioning of the protocol
 - **Users:** use protocol functionality for intended use case
 - **Governance:** make decisions on the development of the protocol
1. For each service, use the stakeholder mapping table to identify who or what the relevant actors are for each category. The more specific, the better. Every category may not be represented, or there may be multiple entries in a category.
 2. Review relevant materials, such as white papers, source code, etc. to identify:
 - a. The specific obligations on each actor
 - b. The specific rewards each actor hopes to receive (in the form of fees, value accrual, categories or other metrics as specified by the protocol).

Complete one stakeholder map per DeFi protocol or service. Blank rows are spaces to add additional stakeholders, where relevant.

Editable versions are available in [Word](#) and [Excel](#) form.

Protocol or service name:	Category	Stakeholders	Responsibility/impact	Economic incentives	Obligations	Rewards
Service category (See Part IC)	Builders	Interface providers	Provide access to DeFi protocols, either directly or through aggregation	Receive transaction fees		
		Auxiliary service providers	Support external data feeds, or offer development tools for DeFi services	Receive transaction fees		
		Connected protocols	Other composable protocols integrated with the target service	Drive utility for their protocol, generate fees		
		Wallet providers	Protect user funds	Fees based on assets		
	Builders and governance	Development teams	Drive development of a protocol and ecosystem	Receive inflationary rewards and transaction fees		
	Governance	Multisig signatories	Shape governance to ensure long-term sustainability	Earn proportion of fees generated by the protocol		
		Governance token holders	Propose and vote on governance decisions	Earn proportion of fees generated by the protocol		
		Miners or stakers	Verify transactions on the underlying blockchain	Receive inflationary rewards and transaction fees		
	Suppliers	Liquidity providers	Contribute collateral or other assets to facilitate DeFi activity	Receive inflationary rewards and transaction fees		
		Liquidators	Liquidate under-collateralized positions	Obtain collateral at discount		
	Users	Protocol users	Use protocol functionality for intended use case	Low-cost, peer-to-peer, trust-minimized financial services		
		Protocol token holders	Use protocol functionality or purchase tokens on secondary markets	Profit from appreciation of token value, or receive inflationary rewards and transaction fees		

Appendix 3:

Decentralization spectrum

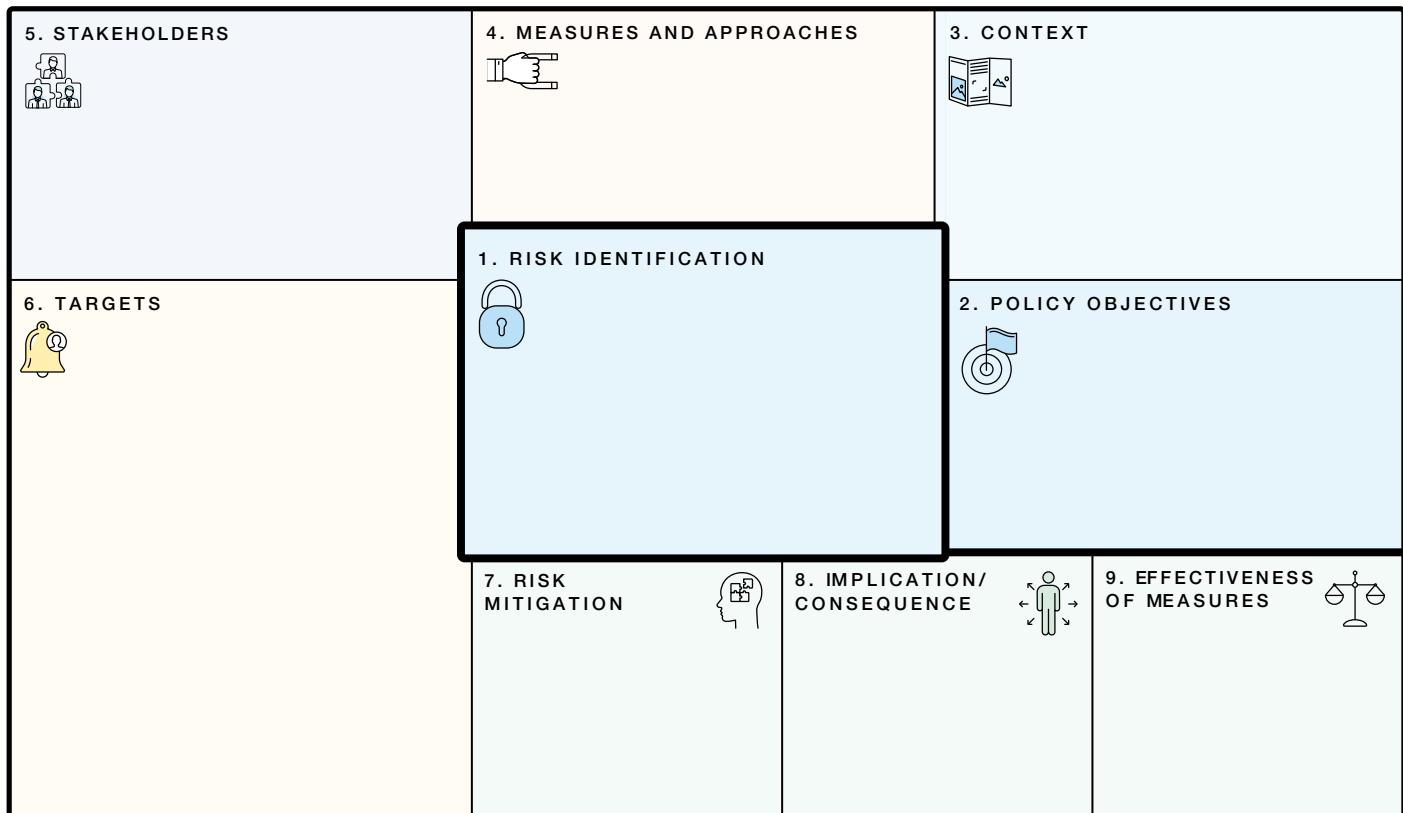
Several aspects of DeFi protocols or services may be more or less decentralized. Furthermore, decentralization can occur at the asset level, at the smart contract level and at the protocol level, to varying degrees.⁴⁵ The following tool maps out the relevant questions to evaluate the spectrum of decentralization in each major area.

	Key questions	Potential spectrum		
Governance	Who decides which aspects of the system can be altered by governance token holders?	Completely centralized	Partially decentralized	Completely decentralized
	What is the threshold to propose governance change?	Only operators can change any aspects of the system	Only some aspects can be altered by governance token holders; threshold for proposing governance change is low	All aspects can be altered, any token holder can propose change
	What percentage of token holders needs to vote on proposal for vote to be valid?			
	Who can vote (all users, all token holders, only governance token holders)?			
	Are all governance tokens freely traded?			
Custody	Who is in charge of safely guarding the assets?	Fully custodial Service retains full control of assets	Partially non-custodial Admin key, time-lock and/or multisig for updating parameters	Completely non-custodial Customer has full control of assets
	Does the user retain control over funds at all times?			
	Who controls the multisignature wallet of the protocol?			
	Are admin keys controlled by a DAO?			
	Are admin keys held in cold storage?			
Protocol modification	Once a smart contract is deployed, can the code be changed by a party unilaterally?	Completely centralized Operators alone can modify all parameters	Partially decentralized Operators can change some parameters; users can change other parameters	Completely decentralized User alone can modify all parameters
	Which parties can make changes to the protocol?			
Verifiable security	Does the development team offer a public bug bounty programme?	No verifiable security Not transparent and unaudited	Some verifiable security Either transparent or audited	Fully verifiable security Formal public verification, with audits from top security firms and a bug bounty programme
	Has there been at least one audit of the code deployed on-chain?			
	Has the audit report been made public?			
	Have all of the serious issues listed in the report been fixed?			
	Have any vulnerabilities been exploited?			
Insurance coverage	Is there insurance coverage? For which risks? Up to what amount?	No coverage Assets are uninsured	Some coverage Limited or non-standardized coverage	Full coverage Assets fully insured
	Is the insurer able to withstand a “black swan event” in DeFi (e.g., substantial coverage claims from different DeFi users simultaneously)?			

Appendix 4:

DeFi policy-maker canvas

The following tool has been developed to help policy-makers frame their consideration of potential approaches to DeFi businesses.⁴⁶ It is designed to apply key components of this toolkit in a structured manner. **Editable versions are available in [Word](#) and [Excel](#) form.**



The canvas is intended to be used working out from the middle counterclockwise, and puts risk identification at its core. The canvas consists of nine questions divided into three stages:

(1) Identifying the necessity and conditions for policy-making

1. *What specific risk are you aiming to address?*
2. *What policy objectives will be achieved by addressing such risk?*
3. *What is the context in which the policy measure will be implemented?*

(2) Defining the approach

4. *What policy measures or approaches are you considering?*
5. *Who are the stakeholders likely to be affected by these measures or approaches?*
6. *Who would be required to take action to implement the measures or approaches?*

(3) Refining the approach

7. Is there already risk mitigation in place (either tech-based or of a self-regulatory nature) and is it sufficient?
8. What would be the implication of this measure, especially regarding innovation, the core business model and Sustainable Development Goals (SDGs)?
9. How effective are these measures, i.e. regarding enforcement?



1. Risk identification

The canvas puts the identified risks at the centre of the policy-making process. As outlined, DeFi may introduce a risk profile different from that presented by conventional financial activities.

As a basis for assessing harms, risks and responses in a structured way, the following questions may be relevant:

- What is the risk and who might suffer harm?
- How significant is the risk to a desired policy outcome?
- Who has a role in reducing/mitigating the risk? What can be done by that entity to mitigate potential harm?
- What legal mechanisms can address that harm?
- How might cross-border activity be addressed?

Example: Operational risks regarding custody (loss of funds)



2. Policy objectives

Before developing a specific approach, policy-makers should identify priorities for policy outcomes. These should serve as a basis for weighing the implications based on proportionality at a later stage (see Refining the approach, above).

Example: Investor protection and financial stability



3. Context

It will be important to understand where DeFi policies and regulations fit within broader regulatory schemes. There may also be existing market structure issues or areas of particular concern in the relevant jurisdiction that bear on decisions concerning DeFi.

Example: National initiatives to promote local development of innovative financial service platforms



4. Measures and approaches

Depending on the layer of the “DeFi stack” addressed, legal mechanisms to address the identified risk of harm will vary. Approaches should be crafted accordingly.

Example: [Gateway] – licensing regime for custodians



5. Stakeholders

This step identifies the groups or categories of individuals who might be affected, positively and negatively, by the proposed measures or approaches.

Example: Investors seeking to leverage their digital assets to increase potential returns; liquidity providers seeking predictable yields for digital assets they hold



6. Targets

This step analyses which actors need to implement the policy or would be encumbered by the measures involved. Activities could be grouped to identify roles, which would then inform specific obligations and controls.

Example: Someone who has control over private keys for others (custodian)



7. Risk mitigation

Policies and regulations should take into consideration existing risk mitigation, which may be tech-based or self-regulatory. It is likely that these will require supplementary measures, but this will give a more informed and nuanced picture of risk.

Example: Self-custody with multisignature wallet and smart contract-enabled governance features (threshold, white-listed addresses, etc.) and auditing of smart contracts



8. Implication/consequence

Desired policy outcomes will need to balance investor protection, innovation and many other considerations. Some measures and approaches will impose significant limitations on DeFi business models. Different levels of impact could be distinguished, for example:

- A low impact if the activity can be conducted without prior approval
- A medium impact if the operation cannot be performed without prior approval
- A high impact if such approval cannot be obtained at all due to the underlying decentralized business model

Example: Licensing regime for custodians = medium impact



9. Effectiveness of measures

As with all policies, the effectiveness of a measure – whether the measure can be enforced and how well it achieves the objective pursued – is an important consideration. Policy-makers and regulators should be clear about how they intend to measure the impact of the policy, weighing both the upsides and downsides, as defined by policy goals and objectives. Key metrics could explore the balance of areas such as consumer protection, privacy, innovation, etc. This also depends heavily on which layer of the tech stack a measure addresses. For instance, policies addressing the network infrastructure layer (blockchain protocol layer) will have more significant and far-reaching implications, and the effects should be measured and considered accordingly.

Example: High effectiveness where regulatory access point can be identified

Contributors

The World Economic Forum’s Centre for the Fourth Industrial Revolution’s work is global, multi-industry and multistakeholder. The project engages stakeholders in various industries and governments from around the world. This report is based on discussions, workshops and research, and the combined effort of all involved. Opinions expressed herein may not necessarily correspond with those of each person involved in the project, nor does it necessarily represent the views of their organizations.

Lead Authors

Sumedha Deshmukh

Platform Curator – Blockchain and Digital Assets, World Economic Forum, USA

André Geest

Researcher, Ludwig Maximilian University, Germany

David Gogel

Growth Lead, dYdX, USA

Daniel Resas

Associated Partner, Schnittker Möllmann Partners, Germany

Christian Sillaber

Senior Researcher, University of Bern, Switzerland

Editor

Kevin Werbach

Professor of Legal Studies and Business Ethics and Director, Blockchain and Digital Asset Project, Wharton School, University of Pennsylvania, USA

The authors would like to thank the following groups and individuals for their insights and contributions:

Content Contributors

Nic Carter, Partner, Castle Island Ventures, USA

Jake Chervinsky, General Counsel, Compound, USA

Tarun Chitra, Chief Executive Officer, Gauntlet, USA

Ann Sofie Cloots, Slaughter & May Lecturer in Company Law, University of Cambridge, United Kingdom

Jacek Czarnecki, Global Legal Counsel, Maker Foundation, Poland

Brendan Forster, Chief Operating Officer, Dharma Labs, USA

Katharina Gehra, Chief Executive Officer, Immutable Insight, Germany

Andreas Glarner, Partner, MME Legal Tax Compliance, Switzerland

Jordan Lazaro Gustave, Chief Operating Officer, Aave, United Kingdom

Siân Jones, Senior Partner, XReg Consulting, Gibraltar

Daniel Kochis, Global Head of Business Development, Chainlink Labs, USA

Joyce Lai, Member, New York Angels; Founder, NewTerritories.io, USA

Urszula McCormack, Partner, King & Wood Mallesons, Hong Kong SAR

Fabian Schär, Professor for Distributed Ledger Technology/Fintech, University of Basel, Switzerland

Lex Sokolin, Head Economist and Global Fintech Co-Head, ConsenSys, United Kingdom

Teana Baker Taylor, General Manager, UK, Crypto.com, United Kingdom

Acknowledgements

Content Reviewers

Marcos Allende Lopez, Technical Leader, LACChain, Inter-American Development Bank, USA
Marvin Ammori, Chief Legal Officer, Uniswap Labs, USA
Sebastian Banescu, Senior Research Engineer, Quantstamp, Germany
Matthias Bauer-Langgartner, Technical Specialist, FCA Innovate, United Kingdom
Nicolas Brügger, Senior Policy Adviser, State Secretariat for International Finance, Switzerland
Jehudi Castro Sierra, Digital Transformation Adviser, Office of the Presidency, Colombia
Charles Dalton, Software Engineer, Blockchain, Crypto, and Digital Currencies, PayPal, USA
Maxim Galash, Chief Executive Officer, Coinchange, Canada
Oli Harris, Vice-President, Goldman Sachs, USA
Kibae Kim, Principal Researcher, Korea Policy Centre for the Fourth Industrial Revolution, KAIST, South Korea
Netta Korin, Co-Founder, Orbs; Founder, Hexa Foundation, Israel
Ashley Lannquist, Project Lead, Blockchain and Digital Currency, World Economic Forum, USA
Caroline Malcolm, Head, Global Blockchain Policy Centre, Organisation for Economic Co-operation and Development, France
Rob Massey, Tax Blockchain, Crypto and Digital Assets Leader, Deloitte, USA
Paul Maley, Global Head of Securities Services, Deutsche Bank, United Kingdom
Xavier Meegan, Blockchain Intern, ING, Netherlands
Michael Mosier, Acting Director, FinCEN, USA
Michael Oh, Director, Office of Financial Innovation, FINRA, USA
Sam Proctor, Chief Executive Officer, Genesis Block, USA
Lane Rettig, Core Team, Spacemesh, USA
Richard Rosenthal, Risk and Financial Advisory, Banking Regulatory Specialist, Deloitte, USA
Lukas Staub, Senior Legal Adviser, State Secretariat for International Finance, Switzerland
Christoph Simmchen, Legal Counsel, Gnosis, Germany
Sheila Warren, Deputy Head, Centre for the Fourth Industrial Revolution, World Economic Forum, USA

Many thanks also to the policy-makers, regulators and industry leaders, both listed and unlisted, who shared their valuable time and insights in roundtables, in individual conversations, and through written feedback.

Editing and Design

Bianca Gay-Fulconis, Designer
Elizabeth Mills, Acting Head of Editing
Alison Moore, Editor

Endnotes

1. We use the general term “digital asset” rather than “cryptocurrency”, “virtual currency” or “cryptoasset”. Particular terms may have distinct legal meanings in certain jurisdictions.
2. The Defiant, “Exclusive: DeFi Year in Review by DappRadar”, 28 December 2020: <https://thedefiant.substack.com/p/exclusive-defi-year-in-review-by-1f2> (link as of 12/5/21).
3. While Bitcoin technically operates on the basis of limited-function smart contracts, Ethereum is a Turing complete blockchain, meaning that it can theoretically support any application that can be executed on a computer.
4. The Defiant, “Exclusive: DeFi Year in Review by DappRadar”, 2020.
5. ConsenSys, *The Q1 2021 DeFi Report*, May 2021: <https://consensys.net/reports/defi-report-q1-2021/> (link as of 12/5/21).
6. Jesse Walden, “How Does DeFi Cross the Chasm?”, 17 June 2020: <https://jessewalden.com/how-does-defi-cross-the-chasm/> (link as of 12/5/21).
7. Linda Xie, “A Beginner’s Guide to DeFi”, 3 January 2020: <https://nakamoto.com/beginners-guide-to-defi/> (link as of 12/5/21).
8. <https://rekt.news/leaderboard/> tracks incidents. One research firm identified nearly \$300 million lost in DeFi hacks between July 2019 and February 2021: <https://cointelegraph.com/news/defi-hacks-and-exploits-total-285m-since-2019-messari-reports>; Chainalysis estimated that \$34 million of DeFi transactions were conducted by criminal actors: <https://coinmarketcap.com/headlines/news/34-million-in-defi-crime-2020/> (links as of 12/5/21).
9. Miles Kruppa and Hannah Murphy, “‘DeFi’ Movement Promises High Interest but High Risk”, Financial Times, 30 December 2019: www.ft.com/content/16db565a-25a1-11ea-9305-4234e74b0ef3 (link as of 12/5/21).
10. Fabian Schär, “Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets”, 2020: <https://ssrn.com/abstract=3571335>; Dirk Zetsche et al., “Decentralized Finance”, Journal of Financial Regulation 6(2), pp. 172–203, 2020: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3539194 (links as of 12/5/21).
11. DeFi market participants often describe services as “protocols” because they are the software code embodied in smart contracts running on the blockchain network. Formally, however, protocols are the technical specification that the software implements.
12. Projects might start out using a centralized implementation with a defined path towards a trust-minimized ecosystem.
13. Products and services offered through permissioned networks and/or blockchains provide additional layers of control and centralization with a distinct risk profile not covered in this report.
14. While Bitcoin is generally not a DeFi foundation because it offers limited smart contract functionality, bitcoin is widely used as a form of collateral for DeFi services, providing bitcoin holders with new options for returns on their holdings.
15. “Custodial” here refers to control over assets, not software code. There may or may not be an entity that has the ability to make unilateral changes to the DeFi protocols or services. Further, we use the term in the colloquial sense of having the ability to move or manipulate assets without the involvement of their owner, not based on any legal definition of “custody” in financial regulation. There are established legal and regulatory requirements governing how customer assets are controlled, such as SEC Rule 15c3-3, which differ from jurisdiction to jurisdiction.
16. European Central Bank, “ESCB/European Banking Supervision Response to the European Commission’s Public Consultation on a New Digital Finance Strategy for Europe/FinTech Action Plan”, 2020:
17. To date, DeFi ecosystems primarily operate on a single blockchain platform. Cross-chain interoperability is the subject of a number of different initiatives.
18. For a similar representation and further analysis, see Schär, “Decentralized Finance”.
19. Sebastian Sinclair, “Uniswap’s First Governance Vote Ends in Ironic Failure”, Coindesk, 20 October 2020: <https://www.coindesk.com/uniswaps-first-governance-vote-ends-in-ironic-failure> (link as of 12/5/21).
20. Some algorithmic stablecoins attempt to maintain sufficient collateralization dynamically as prices shift; others dynamically adjust supply.
21. We use the term “credit” to cover borrowing and lending relationships broadly, rather than in the technical sense of money creation. In contrast to arrangements such as bank loans, in which the borrowing process is separate from the pooling of capital to fund those loans, DeFi services can provide both sides simultaneously, often targeting the same users.
22. Hugh Karp, “Comparing Insurance-Like Solutions in DeFi”, DeFi Prime, 19 November 2019: <https://defiprime.com/comparing-insurance-like-solutions-in-defi> (link as of 12/5/21).
23. It bears noting that many blockchains used for DeFi employ proof-of-stake consensus, which does not involve computationally intensive mining. Ethereum, the blockchain supporting the most DeFi activity, plans to transition to proof-of-stake, as well. Ethereum Foundation, *The Eth2 Upgrades: Upgrading Ethereum to Radical New Heights*: <https://ethereum.org/en/eth2/> (link as of 12/5/21).
24. See case study in Section 2.2.

25. E. J. Spode, "The Great Cryptocurrency Heist", Aeon, 14 February 2017: <https://aeon.co/essays/trust-the-inside-story-of-the-rise-and-fall-of-ethereum> (link as of 12/5/21).
26. William Foxley, "Everything You Ever Wanted to Know About the DeFi 'Flash Loan' Attack", Coindesk 19 February 2020: <https://www.coindesk.com/everything-you-ever-wanted-to-know-about-the-defi-flash-loan-attack>; Yogita Khatri, "Balancer Pools Drained of More Than \$450,000 Due to an Exploit Connected to Deflationary Tokens", The Block, 29 June 2020: <https://www.theblockcrypto.com/linked/69785/balancer-pools-drained-of-more-than-450000-due-to-an-exploit-connected-to-deflationary-tokens>; William Foxley, "Origin Protocol Loses \$7M in Latest DeFi Attack", Coindesk, 17 November 2020: <https://www.coindesk.com/origin-protocol-loses-3-25m-in-latest-flash-loan-attack-reports> (links as of 12/5/21).
27. Connor Sephton, "Overall Losses from Crypto Hacks Are Down – but DeFi Attacks Have Surged", Modern Consensus, 10 November 2020: <https://modernconsensus.com/cryptocurrencies/overall-losses-from-crypto-hacks-are-down-but-defi-attacks-have-surged/> (link as of 12/5/21).
28. Ethereum Foundation, *The Eth2 Upgrades*.
29. C. Sillaber and B. Waltl, "Life Cycle of Smart Contracts in Blockchain Ecosystems". DuD 41, 497–500 (2017). <https://doi.org/10.1007/s11623-017-0819-7> (link as of 28/5/21).
30. Clint Finley, "A \$50 Million Hack Just Showed That the DAO Was All Too Human", Wired, 18 June 2016: <https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/> (link as of 12/5/21).
31. Flashbots, created by researchers in this area, publishes open-source software that miners use for transparent allocation of MEV. However, the approach is controversial; Alex Obadia, "Flashbots: Frontrunning the MEV Crisis", 23 November 2020: <https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752> (link as of 28/5/21).
32. Philip Daian et al., "Flash Boys 2.0: Frontrunning, Transaction Reordering and Consensus Instability in Decentralized Exchanges", IEEE Symposium on Security and Privacy, 2020: <https://arxiv.org/abs/1904.05234> (link as of 12/5/21).
33. Scott Chipolina, "Oracle Exploit Sees \$89 Million Liquidated on Compound", Decrypt, 26 November 2020: <https://decrypt.co/49657/oracle-exploit-sees-100-million-liquidated-on-compound> (link as of 12/5/21).
34. Forks of a decentralized application codebase are distinct from forks of the blockchain settlement layer, such as the split of Ethereum and Ethereum Classic. The forks described here are more analogous to those of other open-source software projects, which create a competing alternative rather than splitting the transaction history.
35. The Defiant, "SushiSwap's Vampire Scheme: Hours Away and with \$1.3B at Stake", 8 September 2020: <https://thedefiant.substack.com/p/sushiswaps-vampire-scheme-hours-away> (link as of 12/5/21).
36. Matthias Nadler and Fabian Schär, "Decentralized Finance, Centralized Ownership? An Iterative Mapping Process to Measure Protocol Token Distribution", IEEE Symposium on Security and Privacy, 2020: <https://arxiv.org/abs/2012.09306> (link as of 12/5/21).
37. Ian Allison, "Inside the Standards Race for Implementing FATF's Travel Rule", Coindesk, 4 February 2020: <https://www.coindesk.com/inside-the-standards-race-for-implementing-fatfs-travel-rule>. FATF draft guidance published in March 2021 suggested that DeFi developers and services might face additional compliance obligations. Nikhilesh De, "State of Crypto: FATF's New Guidance Takes Aim at DeFi", Coindesk, 30 March 2021: <https://www.coindesk.com/fatfs-new-guidance> (links as of 12/5/21).
38. As an early example of such integration, MakerDAO has incorporated a pool of real estate assets into the collateral base for its stablecoin. Muyao Shen, "Maker Price Passes \$4K for First Time, as MakerDAO Brings Real Estate to DeFi", Coindesk, 21 April 2021: <https://www.coindesk.com/maker-price-makerdao-real-world-assets-defi> (link as of 12/5/21).
39. Brady Dale, "Mempool Manipulation Enabled Theft of \$8M in MakerDAO Collateral on Black Thursday: Report", Coindesk, 22 July 2020: <https://www.coindesk.com/mempool-manipulation-enabled-theft-of-8m-in-makerdao-collateral-on-black-thursday-report> (link as of 12/5/21).
40. Appolline Blandin et al., "Global Cryptoasset Regulatory Landscape Study", Cambridge Centre for Alternative Finance (2019), p. 41: https://www.jbs.cam.ac.uk/fileadmin/user_upload/research/centres/alternative-finance/downloads/2019-04-ccaf-global-cryptoasset-regulatory-landscape-study.pdf (link as of 12/5/21).
41. European Commission, "Proposal for a Regulation of the European Parliament and of the Council on Markets in Crypto-Assets", 2019/1937: <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52020PC0593> (link as of 12/5/21).
42. US Securities and Exchange Commission, "SEC Charges EtherDelta Founder with Operating an Unregistered Exchange", news release, 18 November 2018: <https://www.sec.gov/news/press-release/2018-258> (link as of 12/5/21).
43. US Commodity Futures Trading Commission, "Remarks of Commissioner Brian D. Quintenz at the 38th Annual GITEX Technology Week Conference", 16 October 2018: <https://www.cftc.gov/PressRoom/SpeechesTestimony/opaquintenz16>; Aaron Wright and Gary DeWaal, "The Growth and Regulatory Challenges of Decentralized Finance", presentation to the US Commodity Futures Trading Commission Technology Advisory Committee Virtual Currency Subcommittee, 14 December 2020: <https://www.youtube.com/watch?v=qDFXiudwl4&feature=youtu.be> (links as of 12/5/21).
44. US Securities and Exchange Commission, "Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO", release no. 81207, 25 July 2017: <https://www.sec.gov/litigation/investreport/34-81207.pdf> (link as of 12/5/21).
45. Modified from Tony Sheng and Ben Sparango, "Trust Spectrum": <https://multicoin.capital/2020/03/24/trust-spectrum/> (link as of 12/5/21).
46. The DeFi Policy-Maker Canvas is built upon the Digital Policy Model Canvas created by the Transnational Network on National Digital Policy at the World Economic Forum, 15 September 2017.



COMMITTED TO
IMPROVING THE STATE
OF THE WORLD

The World Economic Forum, committed to improving the state of the world, is the International Organization for Public-Private Cooperation.

The Forum engages the foremost political, business and other leaders of society to shape global, regional and industry agendas.

World Economic Forum
91–93 route de la Capite
CH-1223 Cologny/Geneva
Switzerland

Tel.: +41 (0) 22 869 1212
Fax: +41 (0) 22 786 2744
contact@weforum.org
www.weforum.org