Imperial College London

MEng Interim Report

Department of Computing

Imperial College of Science, Technology and Medicine

Solbolt – a compiler explorer and smart contract gas usage tracker for Solidity

Author:
Richard Jun Wei Xiong

Supervisor:
Dr. William Knottenbelt

December 29, 2021

Submitted in partial fulfillment of the requirements for the MEng Computing (Artificial Intelligence and Machine Learning) of Imperial College London

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Planned contributions . . . 3
- 1.3 Outline of report . . . 3

2 Background **4**
- 2.1 The Ethereum Blockchain . . . 4
- 2.2 Smart contracts . . . 5
- 2.3 The Ethereum Virtual Machine . . . 6
  - 2.3.1 Overview . . . 6
  - 2.3.2 Storage . . . 6
  - 2.3.3 Gas consumption . . . 7
- 2.4 Solidity . . . 8
  - 2.4.1 The Solidity language . . . 8
  - 2.4.2 The Solidity compiler and EVM assembly . . . 9
  - 2.4.3 Yul . . . 11

3 Related works **13**

4 Ethical considerations **14**

1

Introduction

> *Whereas most technologies tend to automate workers on the periphery doing menial tasks, blockchains automate away the center. Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly.*
>
> – Vitalik Buterin [1]

1.1 Overview

In recent years, Ethereum has emerged as a promising network for building next-generation, decentralised applications (DApps). Ethereum makes use of blockchain technology to achieve consensus in a distributed, public, and immutable ledger. It supports the running of arbitrary code, called smart contracts, in its own execution environment, called the Ethereum Virtual Machine (EVM). These smart contracts are typically written in a high level programming language, most notably Solidity, and are often used to power the transactional backends of various DApps. They are also used to define the behaviour of utility tokens or various non-fungible tokens (NFTs), which are simply smart contracts that extend a standard interface. Examples would be decentralised exchanges such as Uniswap [2] and Sushiswap, decentralised borrowing and lending platforms such as Aave [3] and Compound [4], algorithmic stablecoins such as Dai [5] and TerraUSD [6], and NFT-powered gaming ecosystems such as Axie Infinity [7] and the Sandbox [8].

Part of the appeal and increasing usage of smart contracts is that it allows transactions and exchanges to take place in a trustless, non-custodial, and cryptographically secure fashion. For example, in the case of a decentralised exchange, there is no need for two untrusted parties to hand over custody of their assets to a trusted intermediary, who would then conduct the actual exchange. Smart contracts are instead able to conduct the exchange of tokens or assets in a single, atomic transaction, which will either succeed, or revert and leave both parties unaffected. This powerful ability

is imparted by Ethereum's immutable and decentralised nature, as no single party can control or alter the state of the blockchain directly.

The result is an explosion in smart contract development and total value locked (TVL), from around $600 million to over $150 billion in the span of two years [9]. However, this has also brought on scalability issues that limit the overall usefulness of the Ethereum network. In simple terms, smart contract transactions each require a certain amount of "gas" in order to be processed, depending on its complexity. Users bid for the price of gas they are willing to pay, in ether, and miners typically choose to process the transactions with higher bids. Since only a limited number of transactions can be processed in each block, this results in an efficient market economy between miners and users.

However, when more users start utilising the network, the gas prices naturally start rising as demand grows. Lately, due to the boom in decentralised finance (DeFi) platforms and NFTs, the gas price has surged by almost 10 times its price compared to two years ago [10]. Transactions are costing up to $63 on average in November [11], and this has made the Ethereum blockchain inhibitive to use for most users [12]. Uniswap, a popular decentralised exchange for trading tokens on the Ethereum network, has spent more than $8.6 million per day in gas fees alone, as reported last November [13]. Therefore, there is a clear need not only for better scaling of the Ethereum blockchain, but also for developers creating DApps on the Ethereum blockchain to optimise their smart contract code for gas usage.

Despite the enormous cost savings that can be achieved via more gas-efficient smart contracts, most of the development on Ethereum smart contract analysis currently surrounds verification of smart contracts rather than gas optimisation [14]. The Solidity compiler itself does offer an estimate for gas usage [15], although only at a course-grained function-call level, making analysis of code within each function difficult. It also only performs a rudimentary best-effort estimate, and outputs infinity whenever the function becomes too complex. Given that Solidity is a language under active development with numerous breaking changes implemented at each major version release [16] [17], the few tools designed with gas estimation in mind are also either no longer maintained or broken, or inadequate for fine-grained, line-level gas analysis. We will discuss some of these existing tools in the background section later.

Currently, most Solidity developers simply make use of general tools such as the Remix IDE, Truffle and Ganache for smart contract development [18]. Such tools are not specialised for detailed gas usage profiling, and developers may potentially miss out on optimisations that may save up to thousands of dollars worth of gas [19], or worse, possibly render a contract call unusable due to reaching the gas limits of each block [20]. Gas optimisation is a step that every developer has to carefully consider when deploying an immutable contract on the blockchain, and therefore creating a tool that calculates the gas usage for each bytecode instruction and line of code could greatly enhance and simplify such analysis, not only for smart contracts on the Ethereum blockchain, but also for all EVM compatible chains such as the

Binance Smart Chain, Polygon, and the Avalanche Contract Chain [21].

1.2 Planned contributions

With the goal of gas optimisation in mind, this project aims to improve on current smart contract development tools by creating a smart contract compiler explorer tool, which can also display a heatmap of the gas usage of each bytecode instruction, while mapping them to the corresponding lines in the user's source code. This is inspired by Godbolt [22], a popular compiler explorer tool for C++ for finding optimisations in the GCC compiler. The following summarises the intended contributions this project plans to achieve.

1. **Compilation and mapping of each line of user Solidity code into their corresponding EVM bytecode and Yul intermediate representation.**

   Similar to Godbolt which is able to easily map and visualise which assembly instructions correspond to its high-level C++ code, we want to achieve a similar result with EVM assembly bytecode and its corresponding Solidity code. We also want to be able to generate the Yul intermediate representation for the Solidity code, which is a recently introduced language between Solidity and EVM assembly.

2. **Bounded static analysis of gas consumption for each EVM instruction**

   The schedule for gas consumption of EVM instructions is carefully laid out in the Ethereum Yellow Paper [23]. Using this, we want to conduct a bounded static analysis of gas consumption for each line of Solidity code, instead of simply at a function call level. To do this, we would need to generate a control flow graph of EVM instructions for each function call, and we plan on making use of user-defined hinted bounds for parameters, such as for variable length stored parameters.

3. **Generation of heatmaps of gas usage per line of code**

   To make visualising gas usage simpler, we plan on generating a heatmap on top of the direct mapping from EVM bytecode instructions to Solidity code, which would display at a glance which lines of code are consuming the most amount of gas, and may potentially have room for optimisation.

4. **Automatic identification of gas inefficient patterns**

   We plan on also identifying some common gas-hungry code patterns, based on previous literature on this topic [19] [24] [25], as well as our own static analysis of the gas usage, and provide suggestions on how to improve them.

1.3 Outline of report

In Chapter

2

Background

2.1 The Ethereum Blockchain

The Ethereum Blockchain can be described as a decentralised, transaction-based state machine [23]. It runs on a proof-of-work system, where, simply put, miners run a computationally difficult algorithm, called Ethash [26], to find a valid nonce for a given block through trial and error. The difficulty of the algorithm is adjusted dynamically to target the mean block time, currently at around 13 seconds [27]. Once a valid nonce is generated, it is very easy for other clients to verify, but almost impossible to tamper with, since changing even one transaction will lead to a completely different hash [28].

Miners are rewarded in Ether with each block they mine, and this gives an economic incentive for more parties to participate in the network. Since the Ethash algorithm is being run by thousands of independent miners concurrently, this makes the Ethereum blockchain more decentralised and secure. A malicious miner will need to be able to consistently solve block nonces faster than other miners to include malicious transactions into each new block, or even reorder blocks [29]. To do this, they would need to control over 51% of the total hashing power of the network, which is incredibly costly and difficult to achieve for a highly decentralised network. This therefore makes the Ethereum blockchain virtually tamper-proof.

The world state of the Ethereum blockchain is a mapping between 160-bit addresses and the corresponding account state. This world state can be altered through the execution of transactions, which represent valid state transitions. Formally:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

where is Υ the Ethereum state transition function, T is a valid transaction, and σ_t and σ_{t+1} are neighbouring states [23]. The main innovation of Ethereum is that Υ allows any arbitrary computation to be executed, and σ is able to store any arbitrary state, not limited to just account balances. This leads us to the concept of smart contracts, which are arbitrary code deployed and stored on the Ethereum blockchain, that can be triggered by contract calls in the form of transactions submitted to the blockchain.

2.2 Smart contracts

The idea of smart contracts was first introduced in 1994 by Nick Szabo [30], where he describes it as a ”computerised transaction protocol that executes the terms of a contract”. Contractual agreements between parties can be directly embedded within the systems that we interact with, which are able to self-enforce the terms of the contract in a way that ”minimise[s] exceptions both malicious and accidental, and minimise[s] the need for trusted intermediaries”, akin to a digital vending machine [31].

The Ethereum blockchain is one of the first widely successful and adopted execution environment for smart contracts. Due to the decentralised and tamper-resistant properties of Ethereum, smart contracts deployed on Ethereum can operate as autonomous actors [32], whose behaviours are completely deterministic and verifiable. Christidis et al also likens them to stored procedures in a relational database [32]. As an example, consider the following smart contract for a simple sale of a token X:

1. Alice wants to sell, for example, 10 X tokens (in this case they could be non-fungible tokens). She writes a buy token function, that accepts 5 Y tokens for each X token sold, and then automatically transfers the correct number of X tokens to the buyer’s address.

2. Bob wants to purchase 2 X tokens. He can then submit a transaction calling the buy token function, with 10 Y tokens from his own account balance. The smart contract will then execute, and Bob will automatically receive 2 X tokens once the transaction is complete. Bob does not need to worry if Alice will honour her part of the contract, because the exchange is atomic and coded within the smart contract itself.

3. Alice can then withdraw the amount of Y tokens she received from the sale, by calling a withdraw profit function. This function would also check that the caller matches Alice’s preset address, so that no one else is able to steal her profit.

4. The smart contract can also keep track of the number of X tokens sold, such that sales will revert after the limit of 10 is reached. It can also automatically revert the transaction if the wrong amount of Y tokens are sent, or even refund Bob the correct amount if he sends too many Y tokens.

Smart contracts can therefore be used to describe any arbitrary contractual agreement, and be used in applications such as voting and governance [33], peer-to-peer marketplaces [34], collateralised borrowing platforms [3] [4], or even NFT ticketing [35] [36]. It is important to note that smart contracts must be completely deterministic in nature, or else each node in the decentralised network will output different resulting (but valid) states.

2.3 The Ethereum Virtual Machine

2.3.1 Overview

Smart contracts on Ethereum run in a Turing-complete execution environment called the Ethereum Virtual Machine (EVM). The EVM runs in a sandboxed environment, and has no access to the underlying filesystem or network. It is a stack machine that has three types of storage – storage, memory, and stack [37], explained in detail in section 2.3.2.

The EVM runs compiled smart contracts in the form of EVM opcodes [38], which can perform the usual arithmetic, logical and stack operations such as `XOR`, `ADD`, and `PUSH`. It also contains blockchain-specific opcodes, such as `BALANCE`, which returns address balance in wei.

An EVM transaction is a message sent from one account to another, and may include any arbitrary binary data (called the *payload*) and Ether [38]. If the target account is a *contract account* (meaning it also stores code deployed on the blockchain), then that code is executed, and the payload is taken as input.

The EVM also supports message calls, which allows contracts to call other contracts or send Ether to non-contract accounts [38]. These are similar to transactions, as they each have their own source, target, payload, Ether, gas and return data.

Each transaction in the EVM can generate any number of logs, which are output by each block. These can be efficiently accessed by off-chain parties, using a bloom filter to find matching logs of a specified event from each block.

2.3.2 Storage

Every account contains a `storageRoot`, which is a hash of the root node of a Merkle Patricia tree that stores the *storage* content of that account [23]. This consists of a key-value store that maps 256-bit words to 256-bit words [38]. This storage type is the most expensive to read, and even more costly to write to, but is the only storage type that is persisted between transactions. Therefore, developers typically try to reduce the amount of storage content used, and often simply store hashes to off-chain storage solutions such as InterPlanetary File System [39] or Arweave permanent storage [40]. A contract can only access its own storage content.

Memory is the second data area where a contract can store non-persistent data. It is byte-addressable and linear, but reads are limited to a width of 256 bits, while writes can be either 8-bit or 256-bits wide. [38] Memory is expanded each time an untouched 256-bit memory word is accessed, and the corresponding gas cost is paid upfront. This gas cost increases quadatically as the memory space accessed grows.

The *stack* is the last data area for storing data currently being operated on, since the EVM is stack-based and not register-based [38]. This has a maximum size of 1024 elements of 256-bit words, and is the cheapest of the three types to access in general. It is possible to copy one of the topmost 16 elements to the top of the stack, or swap

the topmost element with one of the 16 elements below it. Other operations (such as `ADD` or `SUB`) take the top two elements as input, and push the result on the top of the stack. It is otherwise not possible to access elements deeper within the stack without first removing the top elements, or without moving elements into memory or storage first.

2.3.3 Gas consumption

Gas refers to the ”the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network” [41]. It is the fee that is paid in Ether in order to successfully submit and execute a transaction on the blockchain. This mechanism is introduced in order to avoid malicious actors from spamming the network, either by submitting many transactions at once in a denial of service attack, or by running accidental or intentional infinite loops in smart contract code. It also incentivises miners to participate in the network, as part of the gas fees (in the form of tips as introduced in EIP-1559 [42]) is given to the miner .

Each block also has a block limit, which specifies the maximum amount of computation that can be done within each block. This is currently set at 30 million gas units, which is 2 times the target block size [41]. Transactions requiring more gas than the block limit will therefore always revert, which means that all executions will either eventually halt, or hit the block limit and revert.

The schedule for how gas units are calculated is described in detail in Appendix G and H of the Ethereum Yellow Paper [23]. In summary, each transaction first will require a base fee of 21000 gas units, which ”covers the cost of an elliptic curve operation to recover the sender address from the signature as well as the disk and bandwidth space of storing the transaction” [43]. Then, depending on the opcodes of the contract being executed, additional units of gas will be used up. Most opcodes require a fixed gas unit, or a fixed gas unit per byte of data. An exception to this is the gas used for memory accesses, where it is calculated as such [23]:

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

Here, G_{memory} is the gas unit paid for every every additional word when expanding memory. a is the number of memory words such that all accesses reference valid memory. Therefore, memory accesses are only linear up to 724B, after which it becomes quadratic and each memory expansion costs more.

Another exception would be the `SSTORE` and `SELFDESTRUCT` instructions. For these instructions, the schedule is defined as follows [23]:

Name	Value	Description
G_{sset}	20000	Paid for an SSTORE operation when the value of the storage bit is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the value of the storage bit's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into a *refund counter*) when the storage value is set to zero from non-zero.
$G_{\text{selfdestruct}}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
$R_{\text{selfdestruct}}$	24000	Refund given (added into a *refund counter*) for self-destructing an account.

The refund counter tracks the units of gas that is refunded to the user upon successful completion of a transaction, which means that the transaction does not revert or invoke an out-of-gas exception. Only up to half of the total gas used by the transaction can be refunded using the refund counter. This mechanism is introduced to encourage freeing up storage resources used by the Ethereum blockchain.

Therefore, we can see that the estimation of gas usage for any given function is not trivial or easily predictable, and often depends on the current state of the blockchain, as well as the input parameters given. It would also require detailed modelling of the gas and refund counter, as well as other internals of the EVM implementation.

2.4 Solidity

2.4.1 The Solidity language

Solidity is a statically-typed, object-oriented, high-level programming language for developing Ethereum smart contracts [44]. It is a language in active development with numerous breaking changes in each major release, all of which are documented extensively online [15]. We will summarise some of its notable features in this section.

Inheritance. Solidity supports inheritance, by extending other contracts. These contracts can be interfaces (*abstract contracts*), with incomplete implementations of function signatures. A prime example of this would be the ERC-20 and ERC-721 interfaces [45] [46], which are standard interfaces that Ethereum tokens and non-fungible tokens respectively are expected to follow.

Libraries. Solidity also supports the use of libraries, which can contain re-usable functions or structs that are referenced by other contracts. OpenZeppelin for example has a wide range of utility libraries for implementing enumerable sets, safe math, and so on [47].

Function scopes. Solidity allows developers to define the scopes of each function – `internal`, `external`, `private`, or `public` [48]. `internal` functions can only be called by the current contract or any contracts that extend it, and are translated into simple jumps within the EVM. The current memory is not cleared, making such function calls very efficient. `private` functions form a subset of `internal` functions, in that it is stricter by only being visible to the current contract. `external` functions can only be called via transactions or message calls, and all arguments must be copied into memory first. This means that they can only be called "internally" by the same contract via the `this` keyword, which effectively makes an external message call to itself. `public` functions are a superset of `external` functions, in that they can be called internally or externally.

Storage types. Solidity supports the explicit definition of where variables are stored – `storage`, `memory`, `calldata`, or `stack`. `calldata` is an immutable, non-persistent area for storing *function arguments* only, and is reccomended to be used whenever possible as it avoids copies or modification of data. `storage` and `memory` stores the variable in the respective data areas, as described in section 2.3.2.

2.4.2 The Solidity compiler and EVM assembly

The Solidity compiler translates the high-level Solidity code into low-level EVM bytecode, as well as generating other metadata such as the contract Application Binary Interface (ABI), and coarse gas estimates for each function. It supports a built-in optimiser, which takes an `--optimize-runs` parameter. This parameter balances the gas costs of the initial deployment, with the gas costs of future function calls. For example, for a contract that would be called many times after it is deployed, one can set `--optimize-runs=200`, which will generate a larger bytecode, but make future transactions cheaper. The difference that this setting makes in the bytecode may be interesting for developers to examine.

To illustrate the operation of the compiler, let us examine the following example contract, as well as the (truncated) output from the Solidity compiler.

```
1  pragma solidity >=0.5.0 <0.9.0;
2  contract C {
3      function one() public pure returns (uint) {
4          return 1;
5      }
6  }
```

Listing 2.1: An example Solidity contract

```
1  ======= test.sol:C =======
2  ...
3  sub_0: assembly {
4          /* "test.sol":33:123  contract C {... */
5        mstore(0x40, 0x80)
6        callvalue
7  ...
8      tag_1:
9        pop
10       jumpi(tag_2, lt(calldatasize, 0x04))
11       shr(0xe0, calldataload(0x00))
```

```
...
    tag_5:
        /* "test.sol":87:91  uint */
      0x00
        /* "test.sol":111:112  1 */
      0x01
        /* "test.sol":104:112  return 1 */
      swap1
      pop
        /* "test.sol":51:120  function one() public pure returns (uint) {... */
      swap1
      jump  // out
        /* "#utility.yul":7:84   */
...
        /* "#utility.yul":7:84   */
    tag_9:
        /* "#utility.yul":44:51   */
      0x00
        /* "#utility.yul":73:78   */
      dup2
        /* "#utility.yul":62:78   */
...

    auxdata: [...]
}
```

Listing 2.2: EVM assembly from the Solidity compiler

As seen in Listing 2.3, the Solidity code is translated into subroutines with multiple tags as jump destinations, similar to the GCC compiler. It also includes control flow instructions such as `JUMPI` and `JUMP`. Debug information is also embedded into the output, which describes the code fragment that a particular segment of assembly maps to, similar to the `-g` flag in the GCC compiler. These are in the form of character offsets from the top of the source file, rather than line and character numbers.

A detailed explanation of how the EVM assembly maps to the Solidity instructions is included in section 2.4.3, where we examine this in relation to the Yul intermediate representation.

An interesting feature is the `#utility.yul` file that is referenced throughout the output. This file contains extra Yul code (see Section 2.4.3) that are generated automatically by the compiler, and its addition is triggered by the usage of specific language features, such as the ABI encoder for function paramters and return values. They are not specific to the user's source code, but are likely to be executed in control flow jumps from other parts of user code.

The Solidity compiler also has other command-line features as documented online [49], such as the manual linking of library addresses, setting of optimisation options like the inliner or the deduplicator, as well as generation of additional files such as the Abstract Syntax Tree (AST), which can be very useful for the construction of control flow graphs. A recent addition to this is the (experimental) function to generate an intermediate representation of the user Solidity code in Yul, which we will elaborate more in the next section.

2.4.3 Yul

Yul is an intermediate language that is designed to be compiled into bytecode for different backends [50], such as future EVM versions and EWASM, and acts as a common denominator for all current and future platforms. Yul is also designed to be human readable, with high level constructs such as `for` and `switch`, but still suitable for whole-program optimisation.

As an example, we examine the Yul intermediate representation (IR) output from the Solidity compiler, using the same Solidity code in Listing 2.1:

```
1  /// @use-src 0:"test.sol"
2  object "C_10" {
3      ...
4      /// @use-src 0:"test.sol"
5      object "C_10_deployed" {
6          code {
7              ...
8              /// @ast-id 9
9              /// @src 0:51:120  "function one() public pure returns (uint) {..."
10             function fun_one_9() -> var__4 {
11                 /// @src 0:87:91  "uint"
12                 let zero_t_uint256_1 := zero_value_for_split_t_uint256()
13                 var__4 := zero_t_uint256_1
14
15                 /// @src 0:111:112  "1"
16                 let expr_6 := 0x01
17                 /// @src 0:104:112  "return 1"
18                 var__4 := convert_t_rational_1_by_1_to_t_uint256(expr_6)
19                 leave
20
21             }
22             /// @src 0:33:123  "contract C {..."
23
24         }
25        ...
26     }
27 }
```

Listing 2.3: Yul IR from the Solidity compiler

We see that the Yul code far more resembles the original Solidity code. Here, we also observe how the stack is being used in our simple `one()` function:

1. First, a temporary local variable `zero_t_uint256_1` is pushed onto the stack with value `0x00`. Then, it is assigned to the return variable `var__4`. In EVM assembly, this maps to the `0x00` instruction on line 15 of Listing 2.3.
2. Next, another temporary local variable `expr_6` is pushed onto the stack with value `0x01`. It is then assigned to the return variable `var__4` again, and the function exits, returning the value in `var__4`. In EVM assembly, this maps to the `0x01` instruction on line 17. Then, to assign the new value to `var__4`, we first swap it with the old value (currently on the 1st slot in the stack below `0x01`), on line 19. Finally, we pop the old value off the stack, and swap the next jump instruction address onto the top of the stack, in lines 20 and 22.

Similar to the EVM assembly, the generated Yul code also contains debug information about the mapping to the original source code. This can be very useful for

understanding how the final EVM assembly operates, as seen in our previous example. As such, it would be interesting to generate a similar mapping for this project to the Yul intermediate representation, in addition to the final EVM assembly.

3

Related works

4

Ethical considerations

Bibliography

[1] Yoav Vilner. Some rights and wrongs about blockchain for the holiday season, 2018. URL https://www.forbes.com/sites/yoavvilner/2018/12/13/some-rights-and-wrongs-about-blockchain-for-the-holiday-season/. pages 1

[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, Dan Robinson. Uniswap v3 core. URL https://uniswap.org/whitepaper-v3.pdf. pages 1

[3] Aave - protocol whitepaper. URL https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf. pages 1, 5

[4] Robert Leshner, Geoffrey Hayes. Compound: The money market protocol. URL https://compound.finance/documents/Compound.Whitepaper.pdf. pages 1, 5

[5] MakerDAO. The maker protocol: Makerdao's multi-collateral dai (mcd) system. URL https://makerdao.com/en/whitepaper/. pages 1

[6] Evan Kereiakes, Do Kwon, Marco Di Maggio, Nicholas Platias. Terra money: Stability and adoption. URL https://assets.website-files.com/611153e7af981472d8da199c/618b02d13e938ae1f8ad1e45_Terra_White_paper.pdf. pages 1

[7] Axie infinity whitepaper. URL https://whitepaper.axieinfinity.com/. pages 1

[8] The sandbox whitepaper. URL https://installers.sandbox.game/The_Sandbox_Whitepaper_2020.pdf. pages 1

[9] Defi llama: Ethereum tvl. URL https://defillama.com/chain/Ethereum. pages 2

[10] Ethereum average gas price chart. URL https://etherscan.io/chart/gasprice. pages 2

[11] Harry Robertson. Ethereum transaction fees are running sky-high. that's infuriating users and boosting rivals like solana and avalanche. URL https://markets.businessinsider.com/news/currencies/ethereum-transaction-gas-fees-high-solana-avalanche-cardano-crypto-blockchain-2 pages 2

[12] Ishan Pandey. Ethereum's high gas fees is limiting on-chain activities. URL https://hackernoon.com/ethereums-high-gas-fees-is-limiting-on-chain-activities. pages 2

[13] Samuel Wan. Uniswap activity sends ethereum gas fees sky high. URL https://www.newsbtc.com/news/ethereum/uniswap-activity-sends-ethereum-gas-fees-sky-high/. pages 2

[14] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019. doi: 10.1109/DAPPCON.2019.00018. pages 2

[15] *Solidity Documentation Release 0.8.12*, 2021. URL https://buildmedia.readthedocs.org/media/pdf/solidity/develop/solidity.pdf. pages 2, 8

[16] *Solidity v0.7.0 Breaking Changes*, 2021. URL https://docs.soliditylang.org/en/v0.7.0/070-breaking-changes.html. pages 2

[17] *Solidity v0.8.0 Breaking Changes*, 2021. URL https://docs.soliditylang.org/en/v0.8.10/080-breaking-changes.html. pages 2

[18] ConsenSys. *A Definitive List of Ethereum Developer Tools*, 2021. URL https://media.consensys.net/an-definitive-list-of-ethereum-developer-tools-2159ce865974. pages 2

[19] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1433–1448, 2021. doi: 10.1109/TETC.2020.2979019. pages 2, 3

[20] Ethererik. Governmental's 1100 eth jackpot payout is stuck because it uses too much gas, 2016. URL https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/. pages 2

[21] Bitcoin Suisse. Compatible competition - empowering or encroaching on ethereum?, 2016. URL https://www.bitcoinsuisse.com/research/decrypt/compatible-competition-empowering-or-encroaching-on-ethereum. pages 3

[22] Matt Godbolt. Optimizations in c++ compilers. *Commun. ACM*, 63(2):41–49, jan 2020. ISSN 0001-0782. doi: 10.1145/3369754. URL https://doi.org/10.1145/3369754. pages 3

[23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Dec 2021. URL https://ethereum.github.io/yellowpaper/paper.pdf. pages 3, 4, 6, 7

[24] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020. doi: 10.1109/IWBOSE50093.2020.9050163. pages 3

[25] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. 03 2017. pages 3

[26] Ainur R. Zamanov, Vladimir A. Erokhin, and Pavel S. Fedotov. Asic-resistant hash functions. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 394–396, 2018. doi: 10.1109/EIConRus.2018.8317115. pages 4

[27] Vitalik Buterin. Eip-100: Change difficulty adjustment to target mean block time including uncles. Apr 2014. URL https://eips.ethereum.org/EIPS/eip-100. pages 4

[28] Ethereum. Proof-of-work (pow), . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/. pages 4

[29] Pawan Nahar. What are 51 percent attacks in cryptocurrencies?, Aug 2021. URL https://economictimes.indiatimes.com/markets/cryptocurrency/what-are-51-attacks-in-cryptocurrencies/articleshow/85802504.cms. pages 4

[30] Nick Szabo. Smart contracts, 1994. URL https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. pages 5

[31] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sep. 1997. doi: 10.5210/fm.v2i9.548. URL https://firstmonday.org/ojs/index.php/fm/article/view/548. pages 5

[32] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016. doi: 10.1109/ACCESS.2016.2566339. pages 5

[33] Ethereum. Decentralized autonomous organizations (daos), . URL https://ethereum.org/en/dao/. pages 5

[34] OpenSea. Opensea, the largest nft marketplace. URL https://opensea.io/. pages 5

[35] GET Foundation. Get protocol whitepaper. URL https://get-protocol.gitbook.io/whitepaper/. pages 5

[36] Richard Xiong. Tkets: The first decentralised and permissionless nft ticketing platform, powered by theta network. URL https://devpost.com/software/tkets. pages 5

[37] Ethereum. Ethereum virtual machine (evm), . URL https://ethereum.org/en/developers/docs/evm/. pages 6

[38] Solidity. *Introduction to Smart Contracts*, 2021. URL https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html. pages 6

[39] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014. URL http://arxiv.org/abs/1407.3561. pages 6

[40] Samuel Williams, William Jones. Arweave lightpaper, Apr. 2018. URL https://www.arweave.org/files/arweave-lightpaper.pdf. pages 6

[41] Ethereum. Gas and fees, . URL https://ethereum.org/en/developers/docs/gas/. pages 7

[42] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, Abdelhamid Bakhta. Eip-1559: Fee market change for eth 1.0 chain. Apr 2019. URL https://eips.ethereum.org/EIPS/eip-1559. pages 7

[43] Ethereum. Design rationale, . URL https://eth.wiki/en/fundamentals/design-rationale. pages 7

[44] Ethereum. Smart contract languages, . URL https://ethereum.org/en/developers/docs/smart-contracts/languages/. pages 8

[45] Fabian Vogelsteller, Vitalik Buterin. Eip-20: Token standard. URL https://eips.ethereum.org/EIPS/eip-20. pages 8

[46] William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs. Eip-721: Non-fungible token standard. URL https://eips.ethereum.org/EIPS/eip-721. pages 8

[47] OpenZeppelin. Openzeppelin contracts. URL https://github.com/OpenZeppelin/openzeppelin-contracts. pages 8

[48] Solidity. Expressions and control structures, . URL https://docs.soliditylang.org/en/latest/control-structures.html. pages 9

[49] Solidity. Using the compiler, . URL https://docs.soliditylang.org/en/latest/using-the-compiler.html#compiler-input-and-output-json-description. pages 10

[50] Solidity. Yul, . URL https://docs.soliditylang.org/en/latest/yul.html. pages 11