Decentralized Metering and Billing of energy on Ethereum with respect to scalability and security

Aristotle University of Thessaloniki

Honda R&D Europe

Georgios Konstantopoulos

Contents

Abstract

In 2009 Satoshi Nakamoto published the Bitcoin whitepaper [50] where he described 'a purely peer-to-peer version of electronic cash' which 'would allow online payments to be sent directly from one party to another without going through a financial institution'.

Bitcoin was primarily used for fast and low-cost financial transactions which were routed without any bank interference. It was soon realized that its underlying technology, blockchain, could be used for more than transferring financial value. A blockchain is a database that can be shared between a group of non-trusting individuals, without needing a central party to maintain the state of the database. The data in a blockchain is transparent and secured via cryptography. As more advanced blockchain platforms were built ontop of Bitcoin [16], in the end of 2014 a blockchain platform which was capable of executing smart contracts was released, called Ethereum [28]. A smart contract is software that is executed on a blockchain and can be used as framework for secure and trustless computation.

We leverage the power of the Ethereum blockchain and smart contracts to create a system that can transparently and securely perform metering of energy as well as perform accounting for the consumed energy based on specific business logic. The advantages and disadvantages of smart contracts are explored. Due to the distributed nature of blockchains there are challenges towards achieving scalability and achieving transaction throughput comparable to traditional payment processors. Smart contract security has been a pressing issue as large financial amounts have been stolen from smart contracts.

Past literature on current scalability and security issues of smart contracts is studied. Contributions are made on scalability by proposing a method to make data storage on smart contracts more efficient. On security we utilize and augment the functionality of an auditing tool in order to analyze and identify vulnerabilities in smart contracts. Finally, we apply the gained insight and techniques on the metering-billing use case in order to enhance its viability and robustness in production.

1 Introduction

1.1 Problem Statement

The problem this Master Thesis solves is:

> How an organization can manage the energy consumed by a set of energy meters. The system should be able to bill and perform accounting on the metering data, based on a specified accounting model. The system must be transparent, decentralized, and secure. Anyone in the network should be able to verify the validity transactions. Finally, the system needs to scalable at reasonable cost.

1.2 Scope

The Master Thesis explores the fundamental terms needed to understand blockchain terminology. The contributions to scalability are limited to optimizing smart-contracts with respect to not stressing the network. Larger scale scalability solutions such as alternative consensus algorithms, payment channels or sidechains are out of scope. On security, the industry's best practices are applied, while also utilizing tools used by smart contract auditing firms, along with a proprietary tool that was provided for furter analysis.

1.3 Related Work

1.3.1 Scalability

We do not provide contributions towards network level Scalability as it is a far more complex issue than what . To date, there have been proposals [21] which illustrate smart contract techniques that consume less gas and let an application's backend do the heavy lifting. LINK TO PLASMA, TO COSMOS, TO LOOM TO ALL SCALING SOLUTIONS. There is also the case of permissioned blockchains which are able to function with a cryptocurrency that backs them such as Hyperledger Fabric[1] [58].

1.3.2 Security

Tools that are able to analyze and search for vulnerabilities only from contract bytecode have been developed[47, 40, 32]. The recent study[33] on the available tools

[1] `https://www.hyperledger.org/projects/fabric`

that evaluate smart contract security[2] illustrates improvements that can be done to them, and provides an evaluation of each tool. Of these, Oyente and Securify are able to perform direct analysis on a contract's bytecode. Given the contract's source code, Smartcheck is able to vastly outperform other tools, detecting all vulnerabilities proposed in the given taxonomy as well as yielding the least false positives. We provide a rough summary of two tools that are not analyzed in the previous taxonomies or in our Automated Tools section.

These tools utilize symbolic execution. Although useful, raw bytecode analysis is not sufficient here are often false positives and cases where analyzing bytecode is not enough [15].

1.3.2.1 MAIAN

Maian [41] was developed for [40] and is able to detect greedy, prodigal and suicidal contracts which either lock funds indefinitely, leak them to arbitrary users, or are killable by any user MAIAN's features are useful and provide useful insight for creating detection mechanisms that can be incorporated in other tools, however the results of the study are considered skewed for the following reasons[3]:

- All contracts compiled with solc versions earlier than 0.3.6 are considered 'greedy' in absense of a 'withdraw' function due to the fact that functions did not require the 'payable' modifier in order to accept ether.
- Many smart contracts deployed on the Ethereum mainnet are used for testing. When ether was inexpensive, the main network was feasible to be used as its own testnet.
- The only contract which is cited in the paper is one which never received any ether [4]
- No peer-review in the paper

1.3.2.2 Mythril

Mythril [32] is a tool developed by ConsenSys[5]. Its unique feature allows it to directly connect to Ethereum and evaluate a deployed contract at any ethereum network. It allows for direction inspection of contract storage which can be used to evaluate a contract's state. As an example, the aforementioned contract honeypot in Section X could be swiftly inspected and it would be immediately noticed that the 'passHasBeenSet' variable is true.

```
1 $ myth --storage 0,10 -a 0x75041597d8f6e869092d78b9814b7bcdeeb393b4
      --rpc infura-mainnet
2 0x0: 0
    x0000000000000000000000000000000000000000000000000000000000000001
```

[2] Oyente, Remix, SmartCheck, Securify

[3] Opinions influenced by relevant critique on Twitter by TrailOfBits `https://twitter.com/dguido/status/966795086704062465`

[4] `https://etherscan.io/address/0x4671ebe586199456ca28ac050cc9473cbac829eb`

[5] `https://new.consensys.net/`

```
3 0x1: 0
     x262a262b32c44fdf8b75e8180c6bec05873b60a425126bb77e1059b01f6d3513

4 0x2: 0
     x0000000000000000000000000000000000000000000000000000000000000000

5 0x3: 0
     x0000000000000000000000000000000000000000000000000000000000000000

6 0x4: 0
     x0000000000000000000000000000000000000000000000000000000000000000

7 0x5: 0
     x0000000000000000000000000000000000000000000000000000000000000000

8 0x6: 0
     x0000000000000000000000000000000000000000000000000000000000000000

9 0x7: 0
     x0000000000000000000000000000000000000000000000000000000000000000

10 0x8: 0
     x0000000000000000000000000000000000000000000000000000000000000000

11 0x9: 0
     x0000000000000000000000000000000000000000000000000000000000000000
```

1.3.2.3 Solium

Solium [12] is a linter for Solidity. It performs static analysis on the code by analyzing its abstract syntax tree and is able to identify and fix styling and security issues according to preconfigured rules. The plugin for security[6] can be used to further identify security issues in smart contracts according to best-practices as set by the community and by leading companies.

1.3.3 Energy Billing and Accounting on Blockchain

An proposal for billing and accounting models is seen on[57], however it is oriented towards a cloud-management use case and does not implement or go in-depth in a sophisticated smart contract architecture. Attempts to

1.4 Outline

Chapter X describes Y

[6] `https://github.com/duaraghav8/solium-plugin-security`

2 Ethereum and Blockchain Basics

2.1 General Background

Before getting into the specifics of blockchains and Ethereum, the next section will be used to explain fundamental terms on cryptography (hash functions and public key cryptography) and blockchain.

In non technical terms, a blockchain is a database that can be shared by non-trusting individuals without having a central party that maintains the state of the database. Namely, it is a growing list of *blocks* that grows over time. Each block contains various metadata (*blockheaders*) and a number of transactions. A block is chained to its previous one by referencing the previous block's hash. As more blocks get added to the chain, previous blocks and their contents are considered to be more secure.

Any future reference to blockchain terminology such as the contents of a block or a transaction will be referring to the implementations of the Ethereum Platform. The Ethereum Yellowpaper provides details on the formal definitions and contents of each entity [59].

2.1.1 Cryptographic Hash Functions

A hash function is any function that is used to map arbitrary size data to fixed size. The result of a hash function is often called the *hash* of its input. Cryptographic hash functions are hash functions that fullfil certain security properties and are used in cryptography.

More specifically, a secure cryptographic hash function should satisfy the following properties ($H(x)$ refers to the hash of x):

1. **Collision Resistance**: It should be computationally infeasible to find x and y such that $H(x) = H(y)$.

2. **Pre-Image Resistance**: Given $H(x)$ it should be computationally infeasible to find x.

3. **Second Pre-Image Resistance**: Given $H(x)$ it should be computationally infeasible to find x' so that $H(x') = H(x)$. It should be noted that although similar, a second preimage attack on a hash function is significantly more difficult than a preimage attack due to the attacker being able to manipulate only one input of the problem.

Bitcoin uses the SHA-256 cryptographic hash function, while Ethereum uses KECCAK-256. Both functions' outputs are 256 bits long which is considered secure given the document's writing date standards. Ethereum's KECCAK-256 is often referred to as SHA-3 which is inaccurate since SHA3-256 has different padding and thus different values[14].

2.1.2 Public Key Cryptography

Also refered to as Assymetric Cryptography, it is a system that uses a pair of keys to encrypt and decrypt data. The two keys are usually called **public** and **private**[1]. The main advantage of Public Key Cryptography is that it establishes secure communication without the need for a secure channel for the initial exchange of keys between any communicating parties.

The security Public Key Cryptography is based on cryptographic algorithms which are not solvable efficiently due to certain mathematical problems, such as the factorization of large integer numbers for RSA or the discrete logarithm problem for ECDSA[2], being hard.

The three needed properties of secure communication are data integrity, confidentiality and authentication of the sender. We present ways to achieve each of these as follows:

1. **Confidentiality:** By encrypting the plaintext with recipient's public key, the only way to decrypt it is by using the recipient's private key, which is only known to the recipient, thus achieving confidentiality of the message's transmission. This has the disadvantage that it does not achieve authentication and thus anyone can impersonate the sender.

2. **Authentication:** By encrypting the plaintext with sender's private key, the only valid decryption can be done with the sender's public key. This authenticates the identity of the sender of the message. This has the disadvantage that the message can be read by any middle-man as the sender's public key is known.

Achieving both confidentiality and authentication is a two step process. The original message gets encrypted with the sender's private key and encrypted again with the recipient's public key. That way, a recipient decrypts the message firstly with their private key, achieving confidentiality, and then verifies the identity of the sender by decrypting with their private key.

The last part for secure communication is achieving **integrity**. Digital signatures is a scheme which allows the recipient to both verify that the message was created by a sender and that the message has not been tampered with.

The process is as follows:

1. The sender calculates the hash of the message that they are transmitting and concatenates the message with the hash

[1]The public key is a number which is derived by elliptic curve multiplication on the private key. The private key is usually a large number known only to its owner. The public key is in the public domain.

[2]Elliptic Curve Digital Signature Algorithm

2. The sender encrypts the combined message with their private key and transmits the ciphertext to the receiver

3. The receiver decrypts the content of the message with the sender's public key, achieving authentication

4. The receiver hashes the plaintext and compares the result to the transmitted hash

5. If the result matches the transmitted hash then, given that the hashing function used is secure, the message has not been tampered with

If the sender wanted to also make sure of the confidentiality of the information, they'd also encrypt with the receiver's public key after step 2, and similarly the receiver would decrypt with their private key after step 3.

This process is often referred to as a sender broadcasting a *signed* message, due to the usage of this technique.

2.1.3 Advantages of Blockchains

INSERT ADVANTAGES

2.2 Ethereum Blockchain

The Ethereum blockchain acts as a state machine. The first state is the 'genesis' state referred to as the 'genesis block'. After the execution of each transaction, the state changes. Due to the amount of transactions happening in Ethereum, transactions are collated into 'blocks'.

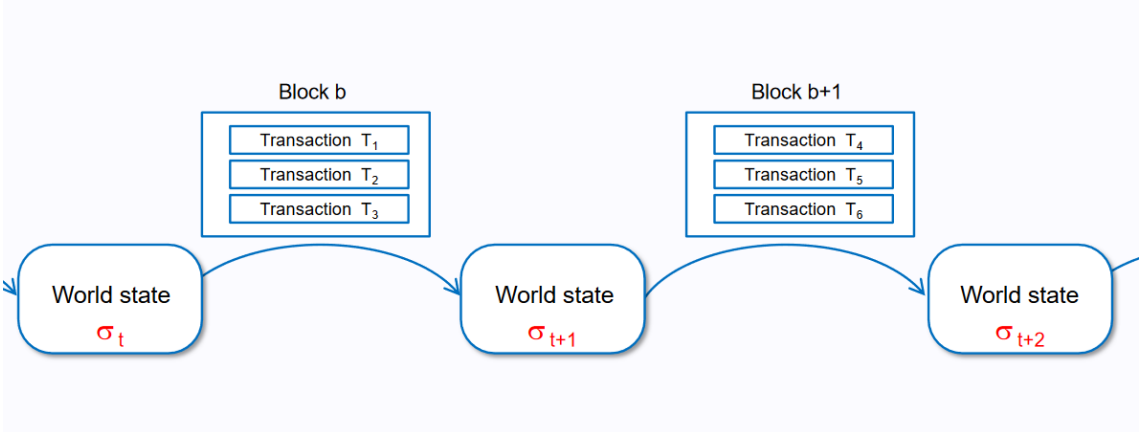


Figure 2.1: Ethereum can be seen as a chain of states, from [55]

A valid state transition requires the appending of a new block to the existing list of blocks. Each block contains transactions and a reference to the previous block, forming a chain. In Ethereum, the only way for a block to be validated and appended to the list is through a validation process called mining. Mining involves a group of computers, known as miners, expending their computational resources to find the solution to a puzzle. The first miner to find a solution to the puzzle is

rewarded with Ether[3] and is able to validate their block proposal. This is a process known as Proof-of-Work (PoW) [34].

Due to having large numbers of miners competing to solve the PoW puzzle, sometimes a miner might solve the PoW at the same time with another miner, but for different block contents. This results in a *fork* of the blockchain. Nodes will accept the first valid block that they receive[4]. Each blockchain implementation has a way to resolve forks and determine which chain is the 'longest'. In Ethereum the longest chain is based on total difficulty[5] which can be found in the blockheader. It should be noted that Ethereum is advertised to be using a modification of the GHOST Protocol[54] as its chain selection mechanism which uses uncle blocks[6]. This contradicts with reality since Ethereun's uncle blocks do not count towards difficulty and as a result, Ethereum does not actually use an adaptation of the GHOST protocol [37]; the uncle reward is just used to reduce miner centralization.

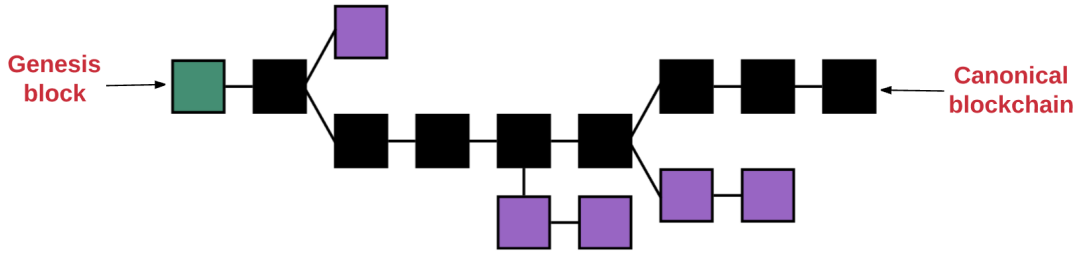


Figure 2.2: Blockchain forks: Ethereum's protocol chooses the canonical chain [44]

2.3 Inside the Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the runtime environment for Ethereum. It is a Turing Complete State machine, allowing arbitrarily complex computations to be executed on it. Ethereum nodes validate blocks and also run the EVM, which means executing the code that is triggered by the transactions. In this section we go over the internals of the EVM.

2.3.1 Accounts

2.3.1.1 World State

Ethereum's global state is a mapping between addresses of accounts and their states. Ethereum full nodes download the blockchain, execute and verify the full result of every transaction since the genesis block. Users should run a full node if they need to

[3]The Ethereum network's native currency

[4]This depends on block propagation time based on bandwith, block-size, connectivity etc.

[5]Difficulty is a measure of how difficult it was for a miner to solve a PoW puzzle. Total Difficulty is the sum of the difficulties of all blocks until the examined block'

[6]In Bitcoin a block with a valid PoW that arrived to a node after another valid block at the same height is called an orphan because it gets discarded by Bitcoin's algorithm. In Ethereum these blocks do not get discarded; instead they are added to the chain as 'uncle blocks' and receive a reduced block reward

execute every transaction in the blockchain or if they need to swiftly query historical data.

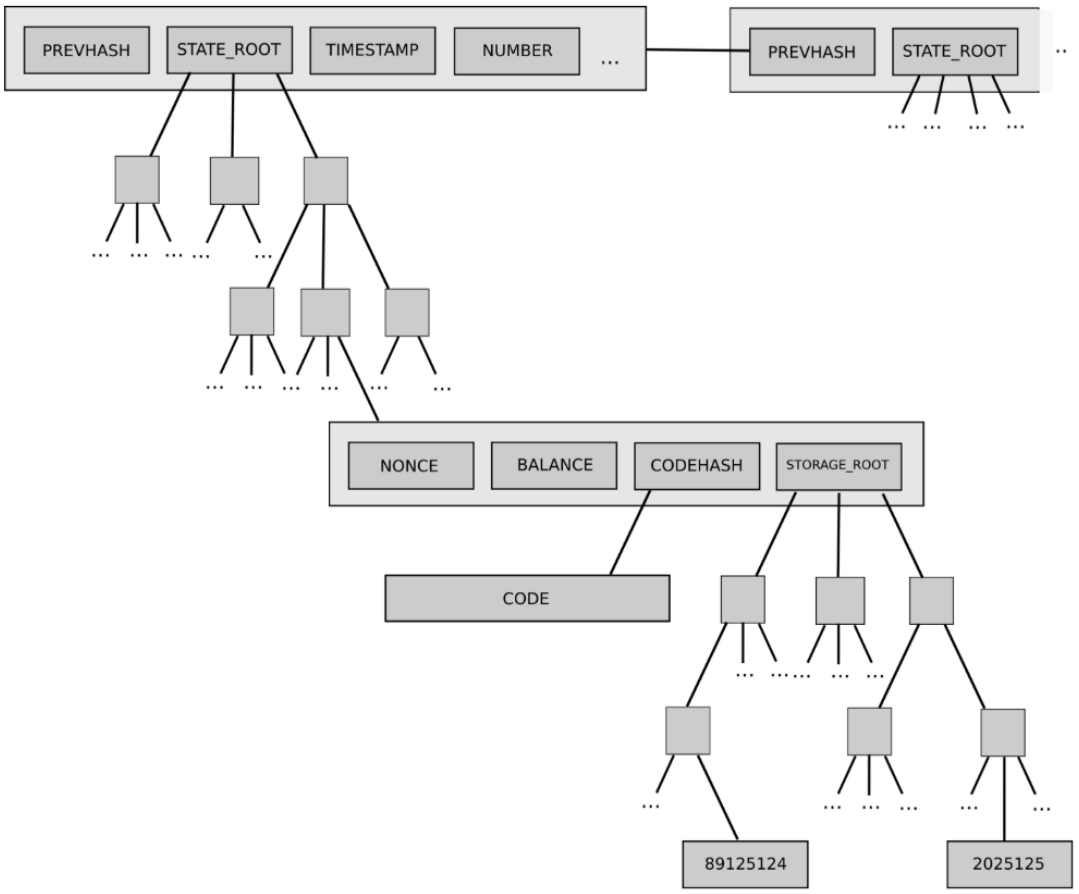


Figure 2.3: The world state of Ethereum

A different kind of node called 'light' node exists for cases where there is no need to store all the information. Instead, light nodes use efficient data structures called *Merkle Trees* which allow them to verify the validity of the data of a tree, even if they don't store the entire tree. A *Merkle Tree* is a binary tree where each parent node is the hash of its two child nodes[7].

[7]Exception: Each leaf node represents the hash of a transaction in a block

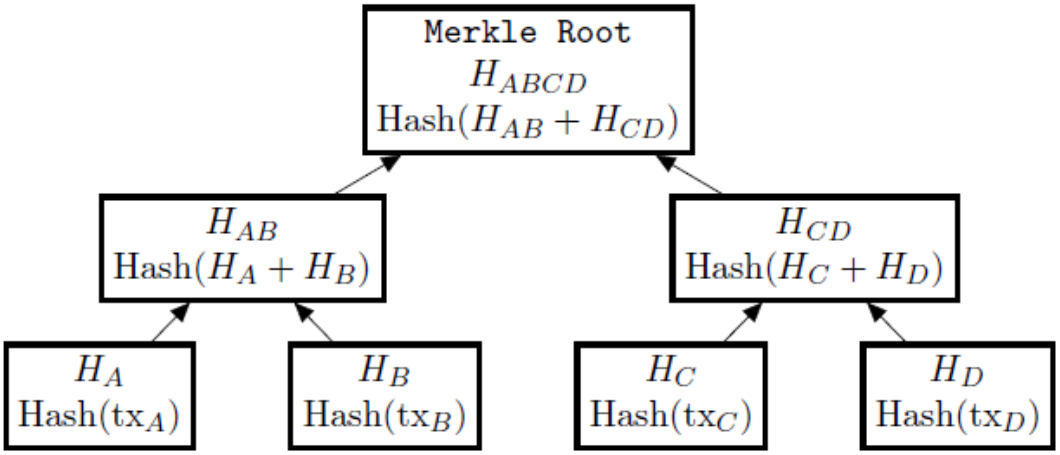
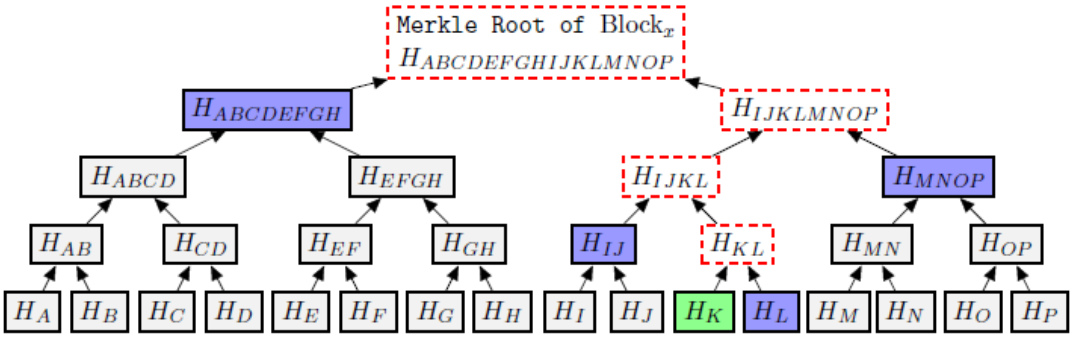


Figure 2.4: Node calculation in a Merkle Tree, from [38]

That way, instead of storing the whole tree of transactions, nodes can verify if a transaction was included in a block or not just by checking if the 'merkle path' to the merkle root is valid. This is efficient as there are only $O(lg_2(n))$ comparisons needed to check the validity of a transaction, as shown in Figure 2.5



:

Figure 2.5: To prove that H_k was included in the merkle root of $Block_x$ only the blue elements are needed, from [38]

2.3.1.2 Account State

An ethereum account is a mapping between an address and an account state. There are two kinds of accounts, Externally Owned Accounts (EOA) and Contract Accounts (CA).

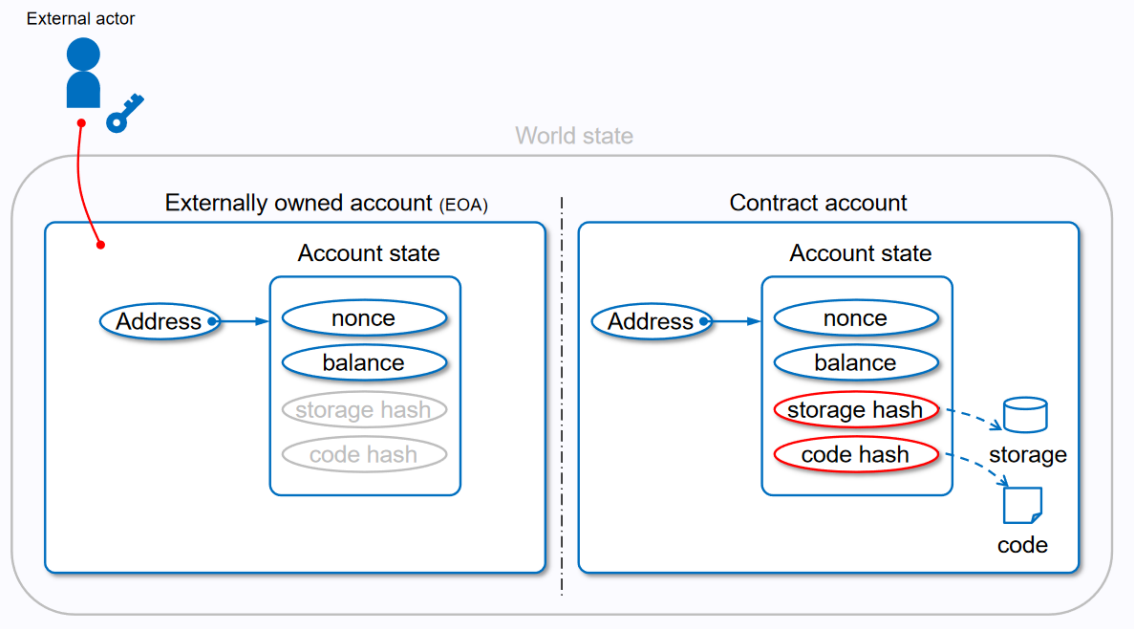


Figure 2.6: EOA is controlled by a Private Key and cannot contain EVM code. CAs contain EVM code and are controlled by the EVM code, from [55]

An EOA is able to send a message to another EOA by signing a transaction with their private key. CAs can make transactions in response to transactions they receive from EOAs.

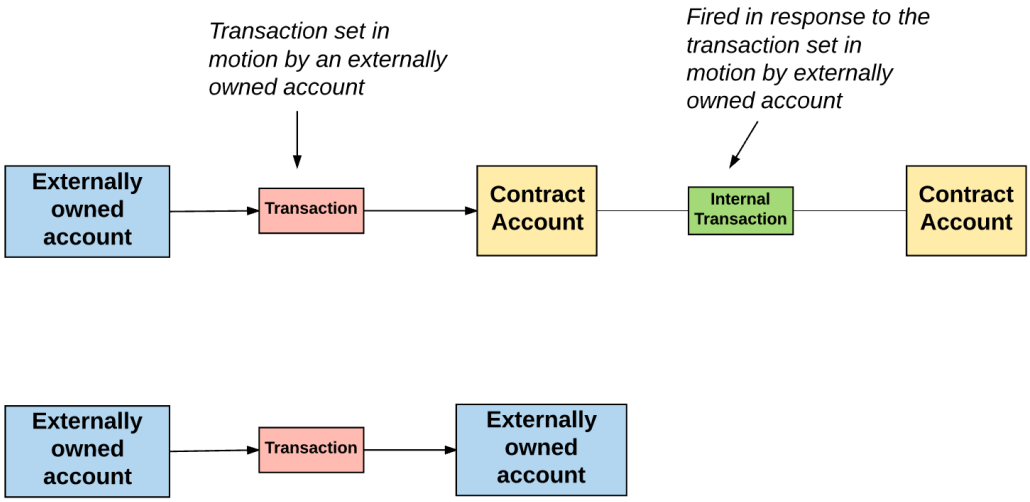


Figure 2.7: EOA can make a transaction to another EOA. A Contract fires a transaction after receiving a transaction from an EOA, from [44]

The public key of an EOA is derived from the private key through elliptic curve multiplication. The address of an EOA is calculated by calculating the KECCAK-256 hash of the public key and prefixing its last 20 bytes with '0x' [59]. The address of a CA is deterministically computed from the sender EOA account's address and their transaction count[8].

[8]Full explanation: `https://ethereum.stackexchange.com/a/761`

We describe the contents of the 'Account State' shown in Figure 2.6 as follows:

1. **Nonce:** The number of transactions sent if it's an EOA, or the number of contracts created if it's a CA.

2. **Balance:** The account's balance denominated in 'wei'[9]

3. **Storage Hash:** The merkle root of the account's storage contents. This is empty for EOAs

4. **Code Hash:** The hash of the code of the account. For EOAs this field is the KECCAK-256 hash of '' while for CAs it is the KECCAK-256 of the bytecode that exists at the CAs address.

2.3.2 Transactions

A transaction is specially formatted instruction that gets signed by an EOA[10] and gets submitted to an Ethereum node. Figure A.1 shows the contents of a transaction as seen after querying an Ethereum node for its contents.

Specifically:

1. **blockHash:** The hash of the block that included the transaction

2. **blockNumber:** The number of the block that included the transaction

3. **from:** The transaction's sender[11]

4. **gas:** The maximum amount of gas that the sender will supply for the execution of the transaction (see 2.3.4)

5. **gasLimit:** The amount of Wei paid by the sender per unit of gas

6. **hash:** The transaction hash

7. **input:** Contains the data which is given as input to a smart contract in order to execute a function. Can also be used to embed a message in the transaction. Contains the value '0x0' in the case of simple transactions of ether.

8. **nonce:** The number of transactions sent by the sender. It is used as a replay protection mechanism.

9. **v, r, s:** Outputs of the ECDSA signature

[9] 1 ether = 10^{18} wei

[10] With the EOAs private key

[11] This field does not actually exist in a transaction however it is recovered from the v,r,s values of the signing algorithm (through 'ecrecover')

2.3.3 Blocks

A block contains the block header and a list of transaction hashes for all the included transactions in that block. Figure A.2 shows the contents of a transaction as seen after querying an Ethereum node for its contents.

Specifically:

1. **difficulty:** The difficulty of the block.

2. **extraData:** Extra data relevant to the block. Miners use it to claim credit for mining a block. In Bitcoin fields with extra data are used to let miners vote on a debate.

3. **gasLimit:** The current maximum gas expenditure per block.

4. **gasUsed:** The cumulative amount of gas used by all transactions included in the block.

5. **hash:** The block's hash.

6. **logsBlom:** A bloom filter which is used for getting further information from the transactions included in the block.

7. **miner:** The address of the entity who mined the block.

8. **mixHash:** A hash used for proving that the block has enough PoW on it.

9. **nonce:** A number which when combined with the mixHash proves the validity of the block.

10. **number:** The block's number.

11. **parentHash:** The hash of the previous block's headers.

12. **receiptsRoot:** The hash of the root node of the Merkle Tree containing the receipts of all transactions in the block .

13. **sha3Uncles:** Hash of the uncles included in the block.

14. **size:** Block size in Kilobytes.

15. **stateRoot:** The hash of the root node of the Merkle Tree containing the state (useful for light nodes).

16. **totalDifficulty:** The cumulative difficulty of all mined blocks until the current block.

17. **transactionsRoot:** The hash of the root node of the Merkle Tree containing all transactions in the block.

2.3.4 Gas

Since all nodes redundantly process all transactions and contract executions this process, this can be used by an attacker to maliciously flood the network with transactions and cause nodes to perform costly computations for extended periods of time. Ethereum uses gas to introduce a cost on performing computations. Gas manifests itself as the fees needed for a transaction (be it value transfer or contract call) to complete successfully.

Every computational step on Ethereum costs gas. The simplest transaction which involves transfering Ether from one account to another costs 21000 gas. Calling functions of a contract involves additional operations wehere the costs can be estimated through the costs described in [19, 59].

When refering to blocks, the gasLimit is the maximum gas that can be included in a block. Since each transaction consumes a certain amount of gas, the cumulative gas used by all transactions in a block needs to be less than gasLimit. There is a similarity between the block gasLimit and the block size in Bitcoin in that they are both used to limit the amount of transactions that can be included in a block. The difference in Ethereum is that miners can 'vote' on the block gasLimit.

Every unit of gas costs a certain amount of *gasPrice* which is set by the sender of the transaction. The cost of a transaction in wei is calculated from the following formula:

$$totalTransactionCost = gasPrice * gasUsed \tag{2.1}$$

Miners are rational players who are looking to maximize their profit. As a result, they include transactions which have higher transaction cost first and transactions with very low transaction fees take longer to confirm.

This effectively creates a fee market where actors increase the gasPrice value to have their transactions confirmed faster. In the times of network congestion such as popular Initial Coin Offerings[12][1] or mass-driven games such as CryptoKitties[13][2] transactions become very expensive, or even taking hours to confirm.

2.3.4.1 Successful Transaction

In the case of a successful transaction, the consumed gas from *gasLimit* goes to miners, while the rest of the gas gets refunded to the sender. After the completion, the world state gets updated.

[12] Crowdfunding for cryptocurrency projects which allow investors to buy tokens in a platform
[13] `https://cryptokitties.co`

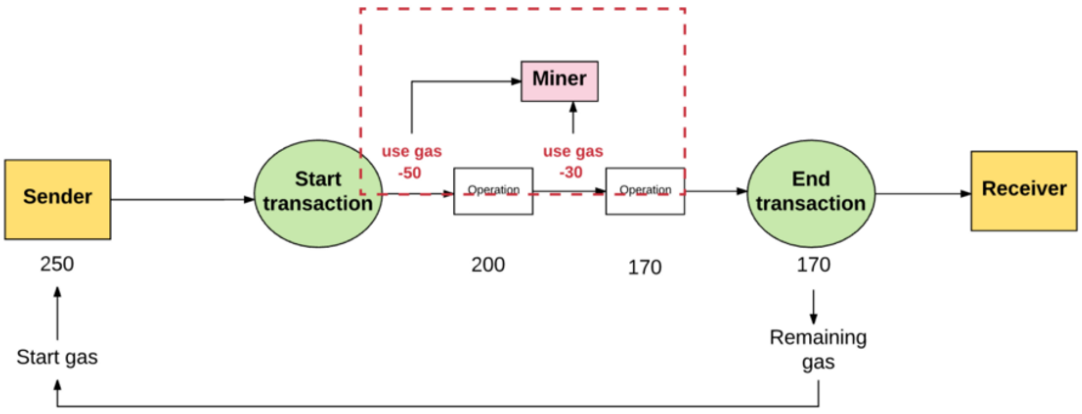


Figure 2.8: Successful transaction, from [44])

2.3.4.2 Failed Transaction

A transaction can fail for reasons such as not being given enough gas for its computations, or some exception occuring during its execution. In this case, any gas consumed goes to the miners and any changes that would happen are reverted. This is similar to the SQL transaction commit-rollback pattern.

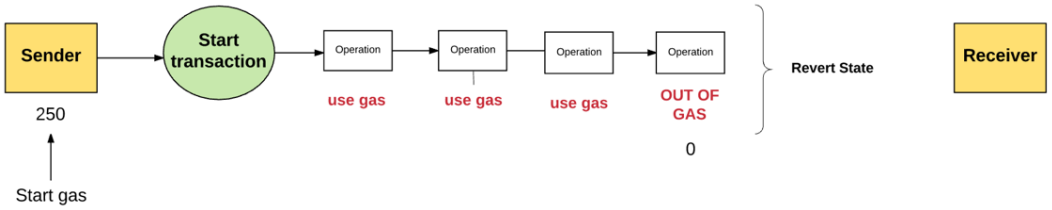


Figure 2.9: Out of gas transaction, from [44])

2.3.5 Mining

The set of rules which allow an actor to add a valid block to the blockchain is called a *consensus algorithm*. In order to have consensus in distributed systems, all participating nodes must have the same version (often called history) of the system (blockchain). A malicious node could create an arbitrary block crediting them with any amount of Ether. In order to avoid that, consensus algorithms elect a network parti:wcipant to decide on the contents of the next block.

Ethereum uses a consensus algorithm called ethash[35] which is a memory-hard[14] consensus algorithm which requires a valid Proof-of-Work in order to append a block to the Ethereum blockchain. PoW involves finding an input called 'nonce' to the

[14]Requires a large amount of memory to execute it. This means that creating ASICs for ethash is harder, although the rumors of an ASIC that can run ethash have been raising discussions on alternatives[3]

algorithm so that the output number is less than a certain threshold[15]. PoW algorithms are designed so that the best strategy to find a valid nonce is by enumerating all the possible options. Finding a valid PoW is a problem that requires a lot of computational power, however verifying a solution is a trivial process, given the nonce. In return, miners are rewarded with 3 ether and with all the fees from the block's transactions.

This process is called mining. In the future, Ethereum is planning to transition to another *consensus algorithm* called Proof of Stake, which deprecates the concept of 'mining' and replaces it with 'staking'. Explaining Proof of Stake and other consensus algorithms is considered out of scope for this Master Thesis.

2.4 Programming in Ethereum

At a low level, the EVM has its own Turing-Complete language called the EVM bytecode. Programmers write in higher-level languages and compile the code from them to EVM bytecode which gets executed in the EVM.

2.4.1 Programming Languages

Programmers can write Ethereum Smart Contracts in languages which have compilers designed to compile to EVM bytecode. Such languages are Solidity, Serpent, LLL or Vyper.

Solidity is the most supported language in the ecosystem and although often comparable to Javascript, we argue that Smart Contracts remind more of C++ or Java, due to their object oriented design. The Solidity Compiler is called 'solc'. In order to deploy a smart contract, its EVM Bytecode and its Application Binary Interface (ABI) are needed, which can be obtained from solc.

```
1  pragma solidity ^0.4.16;
2
3  contract TestContract {
4
5    string private myString = "foo";
6    uint private lastUpdated = now;
7
8    function getString() view external returns (string, uint) {
9      return (myString, lastUpdated);
10   }
11
12   function setString (string _string) public {
13     myString = _string;
14     lastUpdated = block.timestamp;
15   }
16 }
```

Figure 2.10: Basic Solidity Smart Contract

[15]The threshold is also called difficulty and adjusts dynamically so that a valid PoW is found approximately every 12 seconds

Due to the nascence of these languages and the security mistakes that have occured due to them providing programmers with powerful state-changing functions, active research is being done towards safer languages.

2.4.2 Tooling

The following section describes tools and software that are often used by Ethereum users and developers to interact with the network.

2.4.2.1 Client (Node) Implementations and Testnets

Ethereum's official implementations are Geth (golang) and cpp-ethereum (C++). Third party implementations such as Parity (Rust), Pyethereum (Python) and EthereumJ (Java) also exist. The most used kind of node implementations are Geth (compatible with Rinkeby testnet) and Parity (compatible with Kovan testnet).

Smart contracts are immutable once deployed which means that their deployed bytecode (and thus their functionality) cannot change. As a result, if a bug is found when a contract is deployed, the onyl way to fix the bug would be to deploy a new contract. In addition, the deployment costs can be expensive, so development and iterative testing can be costly. For that, public test networks (testnets) exist which allow for testing free of charge. Kovan and Rinkeby are functioning with the Proof of Authority [56] consensus algorithm, compared to Ropsten running Ethash [35] (same as the Ethereum main network but with less difficulty).

Comparison between test networks:

1. Kovan: Proof of Authority consensus supported by Parity nodes only
2. Rinkeby: Proof of Authority consensus supported by Geth nodes only
3. Ropsten: Proof of Work consensus, supported by all node implementations, provides best simulation to the main network

In addition, before deploying to a testnet, developers are encouraged to run their own local testnets in order to further their development processes. Geth and Parity allow for setting up private testnets. Third-party tools also exist that allow for setting up a blockchain with instant confirmation times and prefunded accounts, such as ganache[16] (formerly known as testrpc).

2.4.2.2 Web3

Web3 is the library used for interacting with an Ethereum node. The most feature-rich implementation is Web3.js[17] which is also used for building web interfaces for Ethereum Decentralized Applications (DApps). Implementations for other programming languages are being worked on such as Web3.py[18]. We showcase an example of connecting and fetching the latest block from Ropsten and Mainnet using Web3.js and Web3.py. The full specifications of each library's API can be found in their documentation[19][20]

[16] `http://truffleframework.com/ganache`
[17] `https://github.com/ethereum/web3.js`
[18] `https://github.com/ethereum/web3.py`
[19] `https://github.com/ethereum/wiki/wiki/JavaScript-API`
[20] `https://web3py.readthedocs.io/en/stable/`

2.4.2.3 Truffle

Truffle is the industry standard framework for smart contract development framework written in Node.JS. It allows for easy deployment and initialization smart contracts along with writing test suites utilizing the Mocha testing framework. Latest versions come together with a debugger and a local testnet like ganache.

2.5 Blockchain Types

Blockchains are inspectable and public. Any entity can setup a node, download the full blockchain history and view all the transactions caused by anyone participating in that netwokr. This is one of the main benefits of using a blockchain, transparency. We categorize blockchains in two kinds (different authors might have different classifications):

1. Public or Permissionless: Low barrier to entry, transparent and immutable.
2. Private or Permissioned: Federated participation, can obscure certain pieces of data, ability to modify and revert past transactions if needed.

Vitalik Buterin goes indepth in the advantages and disadvantages between private and public blockchains in [27]. Due to the scalability and privacy restrictions of public blockchains, corporations that are looking to include blockchain technology in their processes are looking for a solution NOW, when the research and development is still not at that level. As a result, permissioned blockchains as JP Morgan's Quorum [49], Hyperledger or Corda have arised, with aims to solve these problems.

3 Blockchain Scalability

3.1 Bottlenecks in Scalability

A blockchain's ability to scale is often measured by the amount of transactions it can verify per second. A block gets appended to the Ethereum blockchain every 12.5 seconds on average, and can contain only a finite amount of transactions. As a result, transaction throughput is bound by the frequency of new blocks and by the number of transactions in them.

We argue that there are two levels of scalability, scalability on contract and on network level. Better contract design can result in transactions which require less gas to execute, and thus allow for more transactions to fit in a block while also making it cheaper for the end user. It should be noted that as Ethereum's current blockGasLimit is set by the miners at 8003916; if all transactions in Ethereum were simple financial transactions[1], each block would be able to verify 381 transactions - 25 transactions per second (tps) - which is still not comparable to traditional payment operators.

3.2 Network Level Scalability

Scale should not be confused with scalability. While scale describes the size of a system and the amount of data being processed, scalability describes how the cost of running the system changes as scale increases. Existing blockchains scale poorly due to the costs associated with them increase faster than the rate at which data can be processed.

First of all, transactions per second as a metric is innacurate. Solving scalability does not imply just increasing the transaction throughput. It is a constraint-satisfaction-problem; the goal is to maximize throughput while maintaining the network's decentralization and security.

[1]Not calls to smart contracts. Transactions without any extra data cost 21000 gas

This sounds like there's some kind of scalability trilemma at play. What is this trilemma and can we break through it?

The trilemma claims that blockchain systems can only at most have two of the following three properties:

- Decentralization (defined as the system being able to run in a scenario where each participant only has access to $O(c)$ resources, ie. a regular laptop or small VPS)
- Scalability (defined as being able to process $O(n) > O(c)$ transactions)
- Security (defined as being secure against attackers with up to $O(n)$ resources)

Figure 3.1: The Scalability Trilemma, from Ethereum's Sharding documentation [10]

As an example that trades decentralization for more transactions is the increase of block size. Increasing the size of each block, implies more disk space for storing the blockchain, better bandwith for propagating the blocks and more processing power on a node to verify any performed computations. This eventually requires computers with datacenter-level network connections and processing power which are not accessible to the average consumer, thus damaging decentralization which is the core value proposition of blockchain.

As described in [52], Proof of Work is a consensus algorithm optimized for censorship-resistance while (in theory) maintain a low barrier to entry. In reality, due to economies of scale, PoW blockchains end up being centralized around small numbers of miners [36].

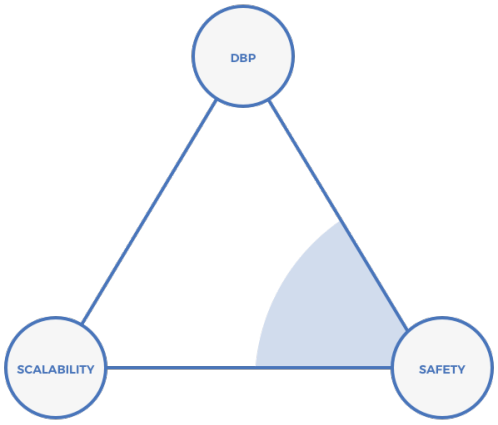


Figure 3.2: Bitcoin and Ethereum's PoW networks have slow probabilistic time to finality and do not scale well. Mining capacity has high concentration in a small amount of entities, from [52]

We proceed to discuss some network level solutions that can improve Ethereum's scalability.

3.2.0.1 Proof-of-Stake

Proof-of-Stake (PoS) is an alternative consensus algorithm where in the place of miners, there are validators who instead of expending computational resources to 'mine' a valid block, they stake[2] their ether and the probability for them to be elected to validate the next block is proportional to their stake. Designing a secure PoS protocol is still under heavy research. The Ethereum Foundation is working on 'Casper the Friendly Finality Gadget'[30] which is a hybrid PoW/PoS consensus algorithm that provides block finality[3] which combined with the 'correct-by-construction Casper the Friendly GHOST'[4] [60] will enable a full transition to Proof of Stake.

3.2.0.2 Sidechains

A sidechain [24] is a blockchain defined by a custom 'rule-set' and can be used to offload computations from another chain. Individual sidechains can follow different sets of rules from the mainchain, which means they can optimize for applications that require high speeds or heavy computation, while still relying on the mainchain for issues requiring the highest levels of security. Ethereum's sidechain solution is called 'Plasma' [51] and involves creating 'child-chains' that run their own consensus algorithm with a two-way peg as described in [24]. *Plasma chains* can have more adjustable parameters such as be less decentralized, however the protocol does not allow for the child-chain operator to abuse their power. A more recent Plasma construct is called 'Plasma-Cash' [29] and describes a more efficient way of executing fraud proofs, in the case of a malicious actor in a *Plasma chain*.

3.2.0.3 Sharding

Due to the architecture of the EVM all transactions are executed sequentially on all ethereum nodes. Sharding refers to splitting the process across nodes, so that each full node is responsible only for a shard[5] and acts as a light client to the other shards. Sharding is the most complex scaling solution and is still at research stages. It also requires a stable Proof of Stake consensus algorithm to function properly.

3.2.0.4 State channels

Contrary to the previous solutions which still record messages on a blockchain, state channels involves exchange of information 'off-chain'. The primary use-case for state channels is micro-transactions between two or more parties. This technique involves exchanging signed messages through a secure communications channel and perform a transaction on the blockchain only when the process is done[6].

[2]Lock up for an amount of time

[3]A block that is finalized cannot be reverted. This is different to traditional PoW which achieves *probabilistic finality*; a block is considered harder to revert the older it is

[4]Uses the GHOST protocol to choose a chain in the case of a fork.

[5]A shard is a part of the blockchain's state

[6]Example: Instead of making 10 transactions worth 0.1 ether each, a transaction is made to open the channel, participants exchange off-chain messages transferring value, and settle or dispute

3.3 Contract Level Scalability

In a recent study [31], after evaluating 4240 smart contracts, it is found that over 70% of them are under-optimized with respect to gas from the compiler. In this section we explore how gas gets computed in smart contracts and ways we can save on gas and transaction costs.

3.3.1 Gas Costs

An Ethereum transaction's gas costs are split in:

1. **Transaction Costs:** The cost of sending data to the blockchain. There are 4 items which make up the full transaction cost:
   (a) The base cost of a transaction (21000 gas)
   (b) The cost of a contract deployment (32000 gas)
   (c) The cost for every zero byte of data in a transaction's input (4 gas per zero byte).
   (d) The cost of every non-zero byte of data in a transaction's input (68 gas per zero byte)
2. **Execution Costs:** The cost of computational operations which are executed as a result of the the transaction, as described in detail in [59, 19]

Gas costs get translated to transaction fees. As a result, a contract should be designed to minimize its operational gas costs in order to minimize its transaction fees. In addition, as gas is a unit for computational costs, less gas consumed results in less burden on the nodes validating the smart contracts which can lead to better scalability.

Table 3.1: Gas cost of different operations, a complete list can be found in Ethereum's yellow paper [19], from [31]

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
AND/OR/XOR	3	Bitwise logic operation
LT/GT/SLT/SGT/EQ	3	Comparison operation
POP	2	Stack operation
PUSH/DUP/SWAP	3	
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional jump
JUMPI	10	Conditional jump
SLOAD	200	Storage operation
SSTORE	5,000/20,000	
BALANCE	400	Get balance of an account
CREATE	32,000	Create a new account using CREATE
CALL	25,000	Create a new account using CALL

As seen in Table 3.1, the most expensive operations involve CREATE[7] and SSTORE operations. The focus of this section will be to explore ways to decrease

the channel with one more transaction at the end.

[7] Used to create a new contract.

gas costs on Smart Contracts, either through better practices or by handcrafting optimizations for specific use cases.

It should be noted, that non-standard methods have been proposed for reducing fees incurred by gas costs. A recent construction [46] describes a method of buying gas at low cost periods and saving it in order to spend it when gas prices are higher[8]. The economic implications of gas arbitrage are outside the scope of this Master Thesis.

General rules that should be followed for saving gas costs:

1. Enable compiler optimizations (although can lead to unexpected scenarios [9]).

2. Reuse code through libraries [42].

3. Setting a variable back to zero refunds 15000 gas through SSTORE, so if a variable is going to be unused it is considered good practice to call 'delete' on it.

4. Use 'bytes32' instead of 'string' for strings that are of known size. 'bytes32' always fit in an EVM word, while 'string' types can be arbitrarily long and thus require more gas for saving their length.

5. Do not store large amounts of data on a blockchain. It is more efficient to store a hash which can be either proof of the existence of the data at a point in time, or it can be a hash pointing to the full data[9]

As described in [31] there is a lot of room for further compiler optimizations. Future Solidity compiler versions are addressing some already[10][11][12]

The EVM operates on 32 byte (256 bit) words. The compiler is able to 'tightly pack' data together, which means that 2 128 bit storage variables can be efficiently stored with 1 SSTORE command. The *optimize* flag of the Solidity compiler needs to be activated to access this feature when programming in Solidity.

3.3.2 Gas Savings Case Study

In order to illustrate our findings and compare across different scenarios, we will perform a Solidity benchmarking test based on a use-case of a Solidity Smart Contract which describes a game. The contract-design requirements are:

- A user must be able to register.

- A user must be able to create a character with certain traits as function arguments.

- A user must be able to retrieve the traits of a character.

[8] When the network is congested
[9] This pattern has been used in combination with IPFS, `https://ipfs.io`
[10] `https://github.com/ethereum/solidity/issues/3760`
[11] `https://github.com/ethereum/solidity/issues/3716`
[12] `https://github.com/ethereum/solidity/issues/3691`

Name	Type	Comment
playerID	uint16	Game supports up to 65535 players
creationTime	uint32	Game supports timestamps up to 2**32 = 02/07/2106 @ 6:28am (UTC)
class	uint4	Game supports up to 16 classes
race	uint4	Game supports up to 16 classes
strength	uint16	Stats can be up to 65535
agility	uint16	Stats can be up to 65535
wisdom	uint16	Stats can be up to 65535
metadata	bytes18	Utilize the rest of the word for metadata

Table 3.2: Required variables and size. Sizes add up to 248 bits which can be efficiently stored in a 256 bit word.

The size of the variables is selected so that all the information required to describe a 'Character' can fit in a 256 bit word. The interface that satisfies the requirements is shown in B.2

For each of the following implementations we will examine the deployment gas costs, as well as the gas costs for calling the 'CreateCharacter' function:

1. Tightly packed structures for setting data
2. Bit masking encoding for setting data
3. Bit masking encoding utilizing libraries, influenced by [53].

The full contracts of each contract can be found in the Appendix. In this section we describe the implementation and in section we go over the results and caveats of each method.

3.3.2.1 Method 1: Tight packing of variables through structs

We're using a solidity 'struct'[13] as a means to group all traits of a character as described in 3.2. This allows for easy code readability since every variable of a struct object can be accessed by its name as seen in B.3c, like the property of an object.

In this case, assignment and retrieval of the variables is done in a very straightforward way. By utilizing Solidity's built-in structures and arrays, we can create an array of 'Character' type structures and access their traits by their indexes, as done in B.3c

The gas costs per function call with this method are:

Optimizer Runs	Register	CreateCharacter	Deployment
0	70003	104205	903173.0
1	69943	104202	529979.0
100	69811	103402	561342.0
500	69604	103207	586867.0
500000	69598	103183	651665.0

Table 3.3: Gas costs for deployment and for each function using Solidity's built-in structs

3.3.2.2 Method 2: Manually pack variables in a 256 bit uint with masking and shifting

In 3.3.2.1 we let the compiler perform optimizations in order to make storing the data more efficient. It turns out[14] that even with the optimizer, a lot of redundant

[13] `http://solidity.readthedocs.io/en/v0.4.21/types.html`

[14] `https://github.com/figs999/Ethereum/blob/master/Solc.aComedyInOneAct`

operations are done. In order to get better results, we create a 'virtual struct' by encoding all the information needed in a 'uint' variable Here we create a new character by shifting variables. Essentially, instead of creating a 'struct' as Solidity expects it and let the compiler do the parsing, we do it ourselves. That way, we achieve gas costs which are substantially lower.

The encoding is done by shifting each of the described variables and then performing bitwise OR operations so that the data gets embedded in the uint variable, as shown in 3.3. This is implemented in B.4

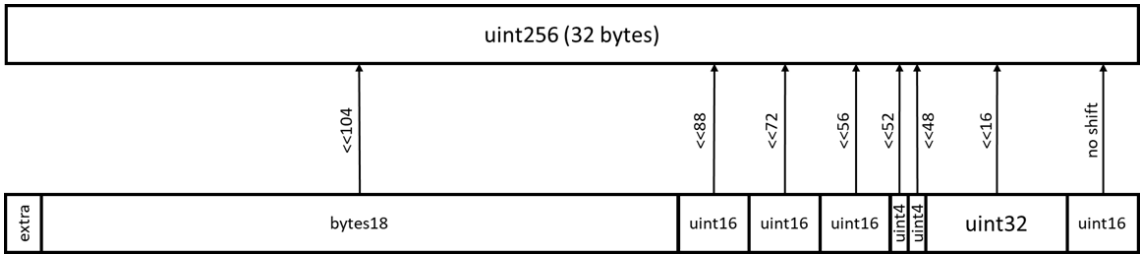


Figure 3.3: Setting data requires shifting left N times and performing bitwise OR with the target variable, where N is the number of bits of the previous variable.

Retrieving the data requires shifting the 'Character' uint and performing bitwise 'AND' operations in order to mask out the higher bits of the variable that are unused, as shown in 3.4, implementation in B.5.

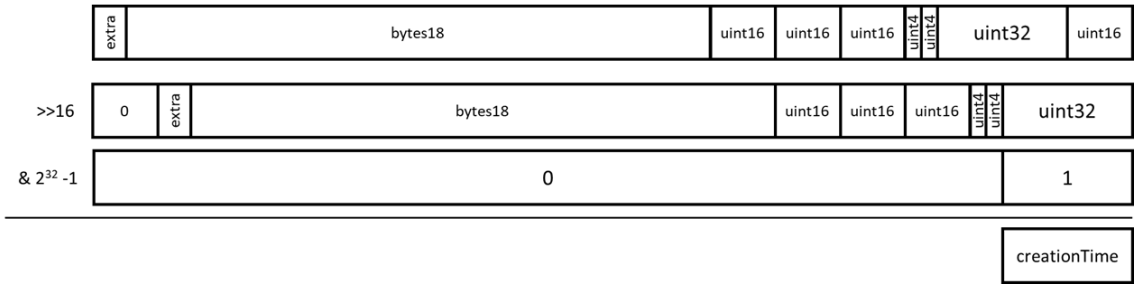


Figure 3.4: Decoding shift and

The gas costs per function call with this method are:

Optimizer Runs	Register	CreateCharacter	Deployment
0	70003	66620	551800.0
1	69943	66365	378022.0
100	69811	65924	402120.0
500	69604	65855	419559.0
500000	69598	65855	432409.0

Table 3.4: Gas costs for deployment and for each function using the masking method on a uint.

3.3.2.3 Method 3: Manually pack variables in a 32 byte variable with masking and shifting, utilizing Solidity Libraries

Code reusability and readability should always be given high priority. In the previous method, although the data packing is very efficient compared to the 'struct' method, the code is hardly readable and it is impossible to reuse parts of it. It would be

possible to extract the logic into functions, however the overheads introduced by the additional operations made it inferior to the technique we describe in this section.

We utilize Solidity's 'Library' feature which allows us to define a set of methods which are possible to be applied on a datatype using the 'using X for Y' syntax[15]

The are two ways to implement functions in a Library:

1. Internal: The library's bytecode is inlined to the main contract's code. This results in larger bytecode during deployment, however each of the contract's function are immediately jumping and returning to the Library's code, like any function. In this case, only the main contract gets deployed.

2. Public:

   (a) The library contract gets compiled and deployed

   (b) The main contract gets compiled and has placeholder slots in the bytecode.

   (c) The placeholder gets replaced by the deployed library's address

   (d) Any function call that requires the library utilizes the 'delegatecall' opcode.

We provide a minimal code snippet which illustrates the syntax to use libraries.

```
1  pragma solidity ^0.4.21;
2
3  library L {
4      function add(uint a, uint b) public pure returns(uint) {
5        return (a+b);
6      }
7  }
8
9  contract C {
10     using L for uint;
11     uint public x = 1;
12     uint public y = 2;
13
14     function add() {
15         x = x.add(y);
16     }
17
18 }
```

Figure 3.5: Example of 'using X for Y' syntax to enhance operations done on datatypes

Libraries with public functions are deployed as standalone contracts to be used by contracts made by other developers. They often include generic functionality such as math operations[16]. Depending on the complexity of the contracts, this can be more efficient compared to using *internal* functions. When *public* is used,

[15] This is similar to calling functions on struct's in Golang

[16] A popular Solidity Library is SafeMath which contains error-checked math operations

the DELEGATECALL opcode is used[17]. When *internal* is used, a function call is interpreted as a jump and thus is more efficient than the DELEGATECALL.

That way, instead of having to deploy a new contract, developers can use an already deployed one. Due to the usage of DELEGATECALL, there is a tradeoff between the deployment cost of the contract and the extra costs when making function calls. We opt for the *internal* approach because it requires less gas and since this is a specialized use-case it is not expected to be used by third-parties.

The final version is split in two files, the library which includes the API for setting and retrieving the character's traits, and the main contract which uses the library's high-level functions.

By utilizing the 'using CharacterLib for bytes32' syntax we are able to write and retrieve the character's traits in a user-friendly manner.

The gas costs per function call with this method are:

Optimizer Runs	Register	CreateCharacter	Deployment
0	70003	67581	754613.0
1	69943	67414	508014.0
100	69811	66904	538621.0
500	69604	66835	556054.0
500000	69598	66835	569032.0

Table 3.5: Gas costs for deployment and for each function using the masking method on bytes32 utilizing Libraries.

3.3.3 Results

It can be seen that in all cases the optimizer's first iteration creates significant gas savings. However, the more optimizer-runs were done, the more gas was spent during deployment. On the other hand, the cost of calling functions went down. Code in Solidity is either optimized for size, and thus costs less to deploy, or for runtime costs[7]. We proceed to compare the gas costs from each implementation while considering the advantages and disadvantages of each method.

In Method 1 (3.3) the optimizer is able to cut down deployment costs by 42%, due to being able to pack all variables in a storage slot. In Method 2 (3.4), the optimization is done manually and has significant results both in deployment and in function costs. In Method 3 (3.5), although we follow the same pattern as in Method 2, we introduce a more verbose API which results in more complex operations being done at a lower level. This introduces slightly bigger deployment costs, and a negligible difference in gas costss for function calls. We consider this overhead to be worth it, since we export an API that is usable.

Deciding how many times to run the optimizer requires estimating the expected usage of the smart contracts. Comparing the results from running the optimizer 1 and 500 times in Method 2, it can be seen that the deployment costs can increase by as much as 10% while the savings for each function call are in the range of 0.7%. When quantifying these percentages in gas, we conclude that if a function is going to be called *enough*[18] times, then the contract which was compiled with more optimizer runs is more efficient in the long run. In addition, this shifts the costs towards the contract deployer rather than the users. Given the tradeoffs described above and

[17]Relays the message to the library while allowing it to execute in the sender's state

[18]Enough is not quantifiable and depends on the implementation. A developer should perform a similar analysis to the one we do in order to set their breakeven points.

the improved code readability and maintainability, we opt to use Method 3 in our implementations.

4 Ethereum and Security

The Ethereum platform itself has proven to be robust and reliable as a blockchain as it has been resistant to both censorship and double-spend attacks. In this chapter we discuss vulnerabilities that have been found in the network's implementation which resulted in Denial of Service-like attacks and the blockchain's state being bloated with junk data. Afterwards, we discuss the security of smart contracts and the best practices that need to be applied in order to have a proper workflow. We contribute to the existing literature by evaluating the usage of the tool 'Slither' towards finding smart contract vulnerabilities and edge cases. We also improved 'Slither' by augmenting the scope of vulnerabilities it was able to detect.

4.1 Past Attacks

4.1.1 Network Level Attacks

October 2016 Spam Attacks During the period of September-October 2016, an attacker was able to spam the Ethereum network's state by creating 19 million 'dead' accounts. The attack was made possible by a mispricing in the SUICIDE opcode of smart contracts, allowing an attacker to submit transactions that created new accounts at a low cost. The creation of these accounts filled the blockchain's state with useless data which resulted in clients being unable to synchronize in time, effectively causing a *Denial of Service* attack to the network [43]. As a response, two hard-forks[1] were proposed [25, 26]. Tangerine Whistle[2] solved the gas pricing issue and at a later point, Spurious Dragon[3] cleared the world state from the accounts created by the attack.

Eclipse Attacks on Ethereum [48] describes *Eclipse* attacks on Ethereum, a type of attack which by flooding a node's TCP connections is able to make them see a different blockchain history than the network's actual one. This was an attack which was known on Bitcoinwhich was considered to be harder to perform on Ethereum nodes. The researchers communicated the potential effects of the attack and the vulnerabilities were fixed in geth v1.8[4]. This vulnerability was not abused in the wild, and as a result there was no need for a hard-fork. It should be noted, that other client implementations such as parity or cpp-ethereum were not found to be

[1] A non-backwards compatible upgrade mechanism that creates new rules for a blockchain, usually to improve the system

[2] EIP608

[3] EIP607

[4] Most popular implementation of Ethereum in golang

vulnerable, which shows that having a diverse set of implementations of a protocol can contribute to the network's security.

4.1.2 Smart Contract Attacks

Contrary to the network itself, Smart Contracts have proven to be quite vulnerable in the past. We proceed to give a brief description and explanation of the three biggest hacks in Ethereum's Smart Contracts, involving the TheDAO and a multisignature wallet implementation by Parity Technologies[5].

4.1.2.1 TheDAO

TheDAO is an acronym for The Decentralized Autonomous Organization. The goal of TheDAO was to create a decentralized business where token holders would vote on porjects to get funded. TheDAO was initially crowdfunded with approximately $150.000.000, the largest crowdfunding in history, to date. In July 2016 it was proven that the smart contract governing TheDAO was vulnerable to a software exploit which enabled an attacker to steal approximately 3.600.000 ether, worth more than $50.000.000 at the time.

If a user did not agree with a funding proposal they were able to get their investment back through a *splitDAO* function in the smart contract. The function was vulnerable to a *reentrancy*[6] attack which allowed an attacker to make unlimited withdrawals from TheDAO contract[18, 17].

What made TheDAO hack very significant was that as a response, part of the Ethereum community decided to perform a hard-fork to negate the mass theft of funds. This was not accepted by the whole community, and as a result, nodes which did not decide to follow the hard-fork, stayed on the original unforked chain which is still maintained and is called 'Ethereum Classic' [4].

4.1.2.2 Parity Multisig 1

In July 2017 a vulnerability was found in the Parity Multisig Wallet[7] which allowed an attacker to steal over 150.000 ether [6]. The attack involved a library contract, which contrary to using Solidity's 'Library' pattern discussed in 3.3.2.3, it involves using the *proxy libraries* pattern [20] to extract the functionality of a smart contract and let it be usable by other contracts, in order to reuse code, and reduce gas costs, as best practices dictate.

The vulnerability in this was that the Library contract involved a *initWallet* function which was being called through the Parity Multisig Wallet. The function was called when the contract was initially deployed in order to set up the owners of the multisig wallet, however due to it being unprotected[8] it was callable by any

[5] `https://www.parity.io/`

[6] Essentially because an account's balance was not reduced before performing a withdrawal it was possible for a malicious user to perform multiple withdrawals and withdraw bigger amounts than their balance allowed.

[7] A cryptocurrency wallet, in this case a smart contract, which requires more than one cryptographic signatures to perform a transaction. It is generally used in organizations and to decrease the chances of funds being stolen.

[8] Public or External visibility without any access control.

user of the wallet[9]. As a result, a malicious user could reinitialize any multisig with their addresss as the contract's owner and drain its funds.

This was observed by a group of hackers called the 'White-Hat Group' who proceeded to drain vulnerable wallets before the attacker could, saving more than $85.000.000 worth of ether at the time. The unrestricted usage of *delegatecall* as well as the lack of proper access control on the *initWallet* function was the root of this hack.

4.1.2.3 Parity Multisig 2

After the first Parity hack, a new multisig wallet library was deployed, with the visibility in the *initWallet* function initialized. This provided the expected functionality to all Parity Multisig implementations which were using the library. The fix in *initWallet* involved adding *only_uninitialized* modifier which would only allow modification of the linked multisig's wallet owners during initialization. However, the library itself was not ever initialized. As a result, any user could call the *initWallet* function and set themselves as the owner of the library contract. This alone would not have been dangerous, had there not been a *kill* function in the smart contract, which when called deletes the contract's bytecode, and effectively renders it useless.

The attacker first[10] became owner of the library by calling *initWallet* with their address as argument, and then proceeded to kill the library. This resulted in **all** contracts that were using the library's logic to be rendered useless, effectively *freezing* 513774 Ether, as well as tokens [8].

A number of proposals were made [13] in order to recover the locked funds. All of these would require an 'irregular state change' similar to what happened with the DAO[11] which was eventually dismissed.

4.2 Evaluating Smart Contract Security

Due to the high financial amounts often involved with smart contracts, security audits from internal and external parties are considered a needed step before deployment to production. It is also being practiced that companies with public smart contracts also engage in bug-bounties, where they encourage users to interact with versions of their contracts deployed on a testnet, in order to identify any other vulnerabilities. Comprehensive studies on identifying the security, privacy and scalability of smart contracts [22] as well as taxonomies aiming to organize past smart contract vulnerabilities have been done [23, 33] have been done, however due to the rapid evolution of the field they get outdated very soon.

There is a need for auditors and developers to use automated auditing tools on their smart contracts and also use the latest version of the Solidity Compiler. As an example, none of the tools mentioned in [33] were able to detect the 'Uninitialized

[9] `https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol`

[10] Although claiming they were not aware of their actions' consequences

[11] 12 million ETH were moved from the "Dark DAO" and "Whitehat DAO" contracts into the WithdrawDAO recovery contract[5]

Storage Pointer' vulnerability[12], however the Solidity Compiler was later updated to throw a Warning if this vulnerability exists.

4.2.1 Automated Tools

Auditing smart contracts significantly more effective when the source code is available. Taking into account the tools which have not been examined in our literature, we came in contact with TrailOfBits, a security auditing firm, and used their suite of tools to extend the already built taxonomies.

We utilized the tool Slither[13] to audit smart contracts which had their source code available. As our concern is primarily in auditing and ensuring smart contracts that have yet to be deployed, we process all the smart contracts with the latest version of the Solidity compiler, v0.4.21, which provides verbose warnings and errors.

As Slither is a static analyzer and works on the source code, its modules (called 'detectors') are to find certain coding patterns which can be considered harmful to the smart contract. This includes detecting popular past contract vulnerabilities such as Reentrancy or the 'Parity bugs', however it's not limited to that. New functionalities can be added through its scriptable API. We describe its modules:

Constant/View functions that write to state: It is planned to make constant and view functions unable to modify state variables by default in the next Solidity compiler versions, however until that happens, it should be enforced manually by developers. It ensures that the code functions as advertised.

Misnamed constructors that allow modification of 'owner'-like variables: A constructor in a smart contract is run once at contract creation and usually sets an 'owner' variable which allows the contract's deployer to have some extra functionality on the contract. In past cases, constructors were not named properly and were callable by adversaries, leading to smart contracts being drained of funds [23]

Reentrancy bugs: After TheDAO brought reentrancy and race-to-empty[14] to the spotlight, all vulnerability scanners for Ethereum smart contracts are able to detect this vulnerability.

Deletion of struct with mapping: Deleting a struct with a mapping inside resets the contents of the struct, however it does not clear the contents of a mapping. This has not been reported as an exploit in the wild[15], however it can be critical in the case of a banking DApp that keeps tracks of balances. A full Proof of Concept is given Appendix A.

Variable Shadowing: This is a unique feature of Slither that has not been implemented in other scanners (has been used in honeypot contracts).

Similar Naming between Variables: Warns users in the case two variables with same length have very similar names. This is used to have more clear variable naming in order to avoid misconceptions and typos.

Unimplemented Function Detection: This ensures that the implementation of an interface stays compliant and does not diverge from the intended specification.

[12] `https://github.com/ethereum/solidity/issues/2628`. This particular vulnerability has been exploited in Smart Contract honeypots as discussed in Section X

[13] Currently not open-sourced. TrailOfBits shared it with us to use it in the thesis.

[14] `http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/`

[15] TrailOfBits have found this bug in audits

Unused State Variables: Detects state variables that are not used in any operations and suggests their removal.

Unprotected Function Detection: Detects public functions which have no modifiers and do not perform any assertions on state variables. The current implementation can impose false positives, however it does not have false negatives. This is able to find the Parity Wallet hack.

Wrong Event Prefix: As per the best practices, the names of 'events' should be capitalized. After a discussion on Github[16], using 'emit' for events is going to be a mandatory for Solidity 0.5.0 and onwards.

It is seen that Slither can be used both for finding known vulnerabilities, but also to avoid common coding anti-patterns and mistakes. Due to its highly scriptable API we can extend it to include more rules. We contributed to the Slither repository by adding support for detecting *tx.origin* and *block.blockhash* usage. The usage of *tx.origin* should be avoided unless necessary, and as stated in the Solidity documentation can incur in loss of funds[17]. *block.blockhash* has been misused in smart contracts and ended up in 400 ETH being stolen from a company called SmartBillions [11]. We also contributed to the improvement of the accuracy of the modules 'UnimplementedFunctionDetection'. Figure X shows a comparison of Slither after our contributions to the other analysis tools from [33].

[CREATE GRAPH WITH SLITHER FINDING VULNS SAME AS OTHER TOOLS]

4.2.2 Honeypot Smart Contracts

Since the second Parity bug and as of March 2018, no novel critical vulnerabilities have been identified in smart contracts. However, smart contracts that are architected to look vulnerable to known exploits started surfacing, when their true purpose is stealing the funds of aspiring hackers. These contract honeypots are funded with an initial small amount of ether (0.5 to 2 ether). Hackers who attempt to exploit them need to first deposit some amount (depending on the honeypot's implementation) before trying to drain the contract. Each honeypot has a well-hidden mechanism to prevent the attacker from draining the funds, essentially locking up any funds that get deposited by individuals other than the contracts deployer.

[16] `https://github.com/ethereum/solidity/issues/2877`
[17] `http://solidity.readthedocs.io/en/v0.4.21/security-considerations.html`

```
1  // contract address: 0xd8993F49F372BB014fB088eaBec95cfDC795CBF6
2  pragma solidity ^0.4.17;
3
4  contract Gift_1_ETH
5  {
6
7      bool passHasBeenSet = false;
8
9      function()payable{}
10
11     function GetHash(bytes pass) constant returns (bytes32) {return
           sha3(pass);}
12
13     bytes32 public hashPass;
14
15     function SetPass(bytes32 hash)
16     payable
17     {
18         if(!passHasBeenSet&&(msg.value >= 1 ether))
19         {
20             hashPass = hash;
21         }
22     }
23
24     function GetGift(bytes pass) returns (bytes32)
25     {
26
27         if( hashPass == sha3(pass))
28         {
29             msg.sender.transfer(this.balance);
30         }
31         return sha3(pass);
32     }
33
34     function PassHasBeenSet(bytes32 hash)
35     {
36         if(hash==hashPass)
37         {
38            passHasBeenSet=true;
39         }
40     }
41 }
```

Figure 4.1: Example honeypot

The above contract was initialized with 1 ether at its balance. An attack can drain the contract by calling the *GetGift* function with the correct password. Due to the attacker not knowing the password, they proceed to change it, using the *SetPass* function, which requires at least a 1 ether deposit, which is acceptable since the attacker will get that back. This also requires that the 'passHasBeenSet' variable is false, or that the PassHasBeenSet function has not been called yet.

A naive attacker would inspect the contract's transactions in Etherscan[18] and after notice that no transaction refering to 'PassHasBeenSet' has been made, and thus proceed to attack the contract and change the password, only to find that

[18]https://etherscan.io/address/0xd8993f49f372bb014fb088eabec95cfdc795cbf6

the password did not get changed. A transaction where a contract calls another contract's function is called a 'Message Call'. Etherscan shows this kind of calls as 'Internal Transactions', only when they include values of more than 0 ether. In this case, 'PassHasBeenSet' does not accept Ether and thus cannot be detected in Etherscan. The contract's owner called 'PassHasBeenSet' from another contract and as a result the password is not changeable. Detecting that the 'passHasBeenSet' variable had been set to true can be done by inspecting the storage of the smart contract, which is always public as shown in C.1

An extensive analysis of smart contracts as honeypots is made in [45] which was released to accompany the research of this Master Thesis.

4.2.3 Towards more secure smart contracts

It is apparent that since smart contracts being unmodifiable after deployment, there is no way of patching any vulnerabilities. Testing platforms have been setup so that developers can practice and test their skills. A developer should keep their code as simple as possible, while providing test coverage for as many scenarios as possible. Using audited and tested code for parts of contracts that have already been implemented (eg. an $ERC20$ token contract) ensures that these parts of the code are going to be secured. Following the best practices as described by the industryś most sophisticated auditors[19]. Finally, developers should be looking for confluence between the results of different automated analyzers in order to filter out false-positives and find false-negatives.

[19] `https://consensys.github.io/smart-contract-best-practices/known_attacks/`

5 Metering and Billing of Energy on Ethereum

5.1 Energy Market inefficiencies

Having discussed how Ethereum works, and explained techniques to improve smart contracts' scalability and security, we proceed to discuss the topic of making energy markets more transparent and efficient by utilizing smart contracts. The use-case we describe can be used as a starting point for better tracking of energy usage inside a company, allowing better prediction of future needs. The world is gradually shifting from nuclear and fossil fuels to Renewable Energy Sources (RES). RES have been taking a larger share of Germany's gross energy production and this has created a Germany is on a rollout plan of installing smart meters in every household which incorporates RES.

The barrier to entry to become an energy producer[1]

In its current state, most consumers do not know what they are paying Business level take long time and are

Price of energy, consumer does not know always what they pay, or what they gain from their renewables

5.2 Advantages of an Energy-based Blockchain application

5.2.1 Peer 2 Peer

In the most general cases, blockchains have the ability to provide transparency and immutability. When talking about energy and transparency, full history of meter readings, price calculation, billing of inhouse energy departments. This can be extended for EV car payment microtransactions and so on.

5.2.2 User Owned Data

5.3 Business Logic

In collaboration with Honda R&D Germany, we create a pilot suite of smart contracts for in-house use in order to track and bill the consumed energy of the com-

[1] By installing solar panels for example

pany's headquarters as measured by a set of smart meters.

Describe meters, billing and so ona The purpose is to serve as a means of tracking the energy consumed by the company's smart meters and ensuring the data's validity and existence in a smart contract. In addition, due to the complex structure of the company, every smart meter's consumption can contribute with different coefficients to the total energy consumption of the rooms in a building. As a result, the developed smart contract are able to track the energy consumption of each room and assign it to a higher-order We proceed to discuss the business logic of the use-case and then implement it. We utilize the technique from 3.3.2.3 to optimize our smart contracts for gas efficiency and utilize Slither from 4.2.1 and the learned best-practices to ensure that the developed smart contracts are robust. Due to the intellectual property of Honda R&D, all testing and verification of the contracts' functionality was done in a private in-house testnet.

5.3.1 Smart Meters and Rooms

A smart meter must be able to keep track of the current reading and timestamp of the reading as well as the last reading and timestamp in order to calculate the difference of the two. It also has a unique identifier which is used to retrieve it in the smart contract.

A company building is split into rooms. Each smart meter contributes to a room's consumption with a real coefficient, according to Equation 5.1

$$R = C * M \tag{5.1}$$

where

$$C = \begin{bmatrix} c_{11} & \cdots & c_{1M} \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ c_{N1} & \cdots & c_{NM} \end{bmatrix}$$

(c_{ij} is the coefficient of the jth meter for the ith room)

and

$$M = \begin{bmatrix} m_1 & \cdots & m_M \end{bmatrix}^T$$

(m_i is the kilowatthour reading of the ith meter)

The coefficients are calculated throuhg an internal partner, INSERT DETAILS ON EASD.

5.3.2 Cost Centers and Billing

Rooms are grouped together in a structure called *Cost Center* which does X. A room can belong only to one cost center. During the accounting stage, the accountant can retrieve the difference in energy consumption since the last clearing period and thus

5.4 Smart Contracts

In this section we go over the implementation and the rationale of each developed smart contract. We explain the inner workings and provide tests of their functionality. A thorough walkthrough on how they interact with each other can be found in 6.1

5.4.1 Contract Registry

Upgradable logic, call smart contract by name.

5.4.2 Meter Management

Meter manager utilizes 3.3.2.3. Each meter has its own ID. We use the pattern. Deleting a meter sets the active status to false. We iterate over the array of meters. There are software engineering patterns [39] that allow more proper usage, however they cost a lot more gas.

5.4.3 Cost - Profit Management

We follow the same pattern as with meters for storing cost centers. We define Follow busienss logic

5.4.4 Access Control

Defining a proper access control policy is very important as discussed in Section 4. It is common to find Access Control Lists (ACL) in enterprise environments which allow access to resources only to selected participants. This does not exist by default in smart contracts. The Aragon Project[2] provides an ACL contract, however it was not used in the final version due to the complexity it introduced to our code[3]. Instead, the DSAUTH pattern is used.

5.5 Monitoring Server

Could be implemented without monitoring server if each meter was smarter. Explain monitoring server

5.5.1 REST API

Explain rest api usage

5.5.2 Python Client

Explain python implementation of rest api

[2]Project aimed at creating DAOs

[3]Aragon's contracts are architected towards creating fully upgradable DAOs, which would introduce considerable overheads and complexity to our code

5.5.3 web3.py interaction

Explain how web3.py interacts with monitoring server and sends data to Smart Contracts

6 Results

6.1 leResults

We are able to create blabla

7 Conclusion

7.1 Future Work

The main issue with our curernt implementation is that instead of having a direct push from each meter (or any IoT device) to the blockchain, we need to pull the data from the aforementioned monitoring server, and then push it again. This introduces latencies and single points of failure, however, this was done due to our corporate setup. An improvement would be to setup a microcontroller on each that would be executing a binary that pings readings to the blockchain. even better, run a node on each IOt device, however requires too much power, maybe in the far future. We explore Ethereum platform due to its abundance in developers and stay in it. There are other smart contract platforms however they lack developer tools, are not battle tested and are potentially centralized. As there is a bigger issue with scaling, the whole infrastructure could be transferred to a permissioned in-house blockchain, however we wanted to stay within the scope of keeping it as transparent as possible.

FIN.

Bibliography

[1] Bat ico, usd 35 million in 24 seconds, gas and gasprice. https://medium.com/@codetractio/bat-ico-usd-35-million-in-24-seconds-gas-and-gasprice-6cdde370a615.

[2] Cat fight? ethereum users clash over cryptokitties. https://www.coindesk.com/cat-fight-ethereum-users-clash-cryptokitties-congestion/.

[3] Eip: Modify block mining to be asic resistant. https://github.com/ethereum/EIPs/issues/958.

[4] Ethereum classic. https://ethereumclassic.github.io/.

[5] Hard fork completed.

[6] An in-depth look at the parity multisig bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[7] Optimizer seems to produce larger bytecode when run longer. https://github.com/ethereum/solidity/issues/2245.

[8] A postmortem on the parity multi-sig library self-destruct. http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[9] Psa: Beware of buggy solidity version v0.4.5+commit.b318366e - it's actively used to try to trick people by exploiting the mismatch between what the source code says and what the bytecode actually does. https://www.reddit.com/r/ethereum/comments/5fvpjq/psa_beware_of_buggy_solidity_version/.

[10] Sharding faq. https://github.com/ethereum/wiki/wiki/Sharding-FAQ.

[11] Smartbillions challenges hackers with 1,500 ether reward, gets hacked and pulls most of it out. urlhttps://www.ccn.com/smartbillions-challenges-hackers-1500-ether-reward-gets-hacked-pulls/.

[12] Solium. Lintertoidentifyandfixstyle&securityissuesinSolidity.

[13] Standardized ethereum recovery proposals. urlhttps://github.com/ethereum/EIPs/pull/867.

[14] Which cryptographic hash function does ethereum use? https://ethereum.stackexchange.com/a/554.

[15] Zeus: Analyzing safety of smart contracts.

[16] Colored coins, 2013.

[17] Analysis of the dao exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016.

[18] Ether thief remains mystery after $55 million heist. https://www.bloomberg.com/features/2017-the-ether-thief/, 2017.

[19] Gas costs from yellow paper – eip-150 revision (1e18248 - 2017-04-12). https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0, 2017.

[20] Proxy libraries in solidity. https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd, 2017.

[21] Stateless smart contracts. https://medium.com/@childsmaidment/stateless-smart-contracts-21830b0cd1b6, 2017.

[22] Maher Alharby and Aad van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *CoRR*, abs/1710.06372, 2017.

[23] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[24] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[25] Alex Beregszaszi. Hardfork meta: Spurious dragon. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-607.md.

[26] Alex Beregszaszi. Hardfork meta: Tangerine whistle. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-608.md, 2017.

[27] Vitalik Buterin. On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains, 2015.

[28] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[29] Vitalik Buterin, Karl Floersch, and Dan Robinson. Plasma cash: Plasma with much less per-user data checking, 2018.

[30] Vitalik Buterin and Griffith Virgil. Casper the friendly finality gadget, 2017.

[31] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. *CoRR*, abs/1703.03994, 2017.

[32] ConsenSys. Mythril. https://github.com/ConsenSys/mythril.

[33] Ardit Dika. Ethereum smart contracts: Security vulnerabilities and security tools, 2017.

[34] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[35] Ethereum. Ethash. https://github.com/ethereum/wiki/wiki/Ethash.

[36] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. *CoRR*, abs/1801.03998, 2018.

[37] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 3–16, New York, NY, USA, 2016. ACM.

[38] Erik Hilbom and Tillstrom Tobias. Applications of smart-contracts and smart-property utilizing blockchains, 2016.

[39] Rob Hitchens. Solidity crud. https://bitbucket.org/rhitchens2/soliditycrud.

[40] Nikolic Ivica, Kolluri Aashish, Sergey Ilya, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale, 2018.

[41] Nikolic Ivica, Kolluri Aashish, Sergey Ilya, Prateek Saxena, and Aquinas Hobor. Maian. https://github.com/MAIAN-tool/MAIAN, 2018.

[42] Jorge Izquierdo. Library driven development in solidity. https://blog.aragon.one/library-driven-development-in-solidity-2bebcaf88736, 2017.

[43] Hudson Jameson. Faq: Upcoming ethereum hard fork. https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/, 2016.

[44] Preethi Kasireddy. How does ethereum work, anyway? 2017.

[45] Georgios Konstantopoulos. Hacking the hackers: Analyzing smart contract honeypots, 2018.

[46] Florian Tramèr Lorenz Breidenbach, Phil Daian. Tokenize gas on ethereum with gastoken. https://gastoken.io, 2018.

[47] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[48] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. Cryptology ePrint Archive, Report 2018/236, 2018. https://eprint.iacr.org/2018/236.

[49] J.P Morgan. A permissioned implementation of ethereum supporting data privacy. https://www.jpmorgan.com/country/DE/en/Quorum.

[50] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[51] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017.

[52] Kyle Samani. Models for scaling trustless computation. https://multicoin.capital/2018/02/23/models-scaling-trustless-computation/.

[53] Chance Santana-Wees. Virtualstruct.sol. https://github.com/figs999/Ethereum/blob/master/VirtualStruct.sol.

[54] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin, 12 2013.

[55] Takenobu T. Ethereum virtual machine illustrated. http://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

[56] Parity Technologies. Proof-of-authority chains. https://wiki.parity.io/Proof-of-Authority-Chains.html.

[57] Mukesh Thakur. Authentication, authorization and accounting with ethereum, 2017.

[58] Marko Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, pages 3–7, New York, NY, USA, 2017. ACM.

[59] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[60] Vlad Zamfir. Casper the friendly ghost: A "correct-by-construction" blockchain consensus protocol, 2017.

List of Figures

List of Tables

Appendices

A Transactions and Blocks

Querying a node for transactions:

```
> web3.eth.getTransaction("0
    x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
    ")
{ blockHash: '0
    x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
    ',
  blockNumber: 5284738,
  from: '0xbc7f7a5eb2039a3330665aa22b5d79485896bd11',
  gas: 250000,
  gasPrice: BigNumber { s: 1, e: 10, c: [
      80000000000
    ]
  },
  hash: '0
     x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
     ',
  input: '0
     x0a19b14a0000000000000000000000000000000000000000000000000000000000000000
     ',
  nonce: 9,
  to: '0x8d12a197cb00d4747a1fe03395095ce2a5cc6819',
  transactionIndex: 0,
  value: BigNumber { s: 1, e: 0, c: [
      0
    ]
  },
  v: '0x1c',
  r: '0
     xe7df9fee346ea3575dd541e120df7b154d46bba2bf859352b29a7f2c6dd5fab3
     ',
  s: '0
     x68d266b140ac468ee328592ca6b4a461edace859d6837358bc3f8eeeb24ade63
     '
}
```

Figure A.1: Contents of an Ethereum transaction when querying a node

Querying a node for block contents:

```
> web3.eth.getBlock(5284738)
{ difficulty: BigNumber { s: 1, e: 15, c: [
      32,
      85319757566868
    ]
  },
  extraData: '0x7869786978697869',
  gasLimit: 7995219,
  gasUsed: 1547361,
  hash: '0
     x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
     ',
  logsBloom: '0
     x0002080000020400020004000000000000010040000000000800020000000084000800400220
     ',
  miner: '0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb',
  mixHash: '0
     x29b6efa55ad0298b0c90f21e9e23d572977ffb3c5064a9816a69bb2bf2a9effd
     ',
  nonce: '0xabed128000fed25e',
  number: 5284738,
  parentHash: '0
     xb7063b9c7b05c95c35a329717e44875829cc740b2e0749e03d54806dcf34b520
     ',
  receiptsRoot: '0
     xe5e176557b9f40394917191095b706a2a331742f0dc93a10e1d59b5e297ee0b5
     ',
  sha3Uncles: '0
     x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
     ',
  size: 7789,
  stateRoot: '0
     x1c62917ac72a2b76e00053efbb7af0d6949e86cafb3f983812d763715c6c9905
     ',
  timestamp: 1521484243,
  totalDifficulty: BigNumber { s: 1, e: 21, c: [
      31406307,
      78318927526632
    ]
  },
  transactions: [ '0
     x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
     ',
     '0
        xbe1c3e767e34d5d668ea50d3400b2e11a663479f931c225eda5e1d314e012589
        ', ...
  ],
  transactionsRoot: '0
     xb0a066469d74fe1f450c5fa8a1f59c5b7305feb6336d0d59f347a2b2c7a8c579
     ',
  uncles: []
}
```

Figure A.2: Contents of an Ethereum block when querying a node

Ganache UI:

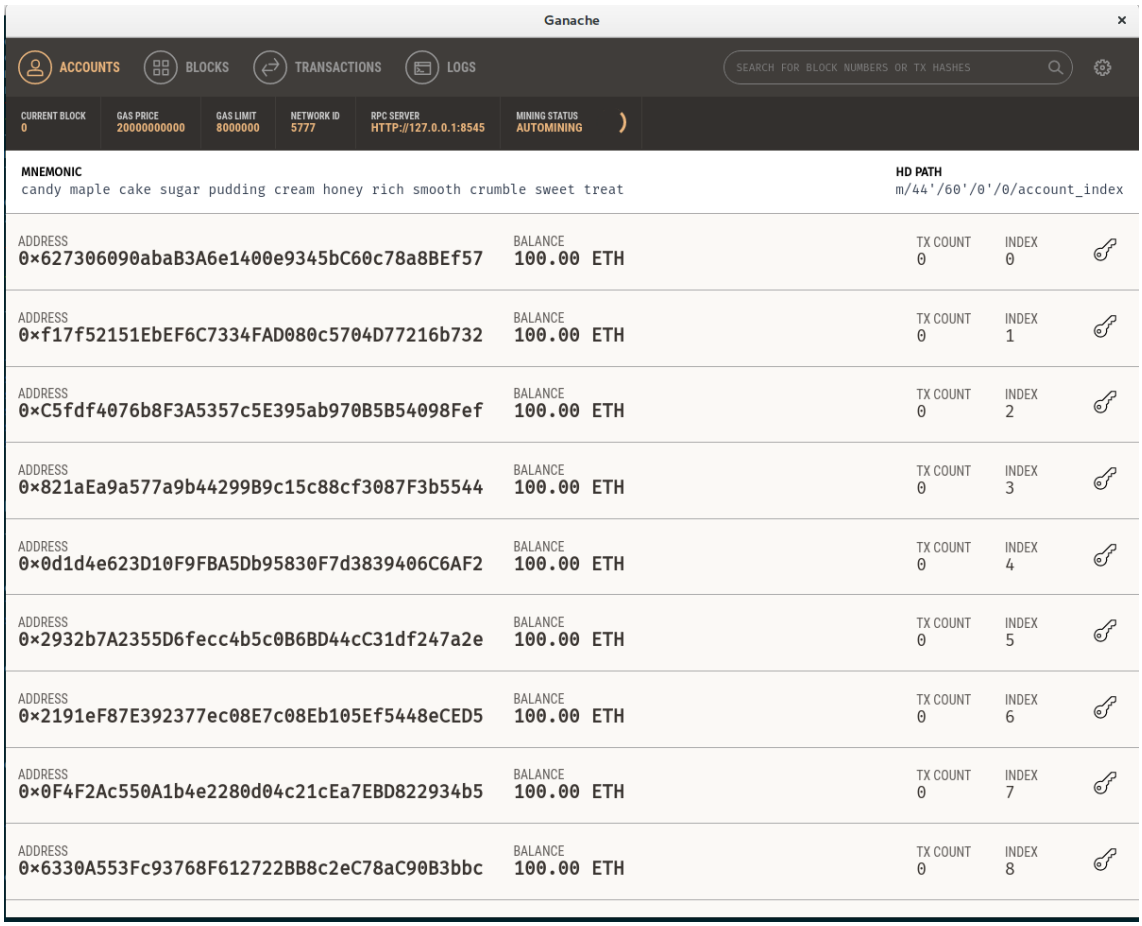


Figure A.3: Ganache testnet User Interface

Web3js example:

```
node
     > Web3 = require('web3');
> INFURA_API = process.env.INFURA_API; // Infura is a third party
    service that allows us to connect to their Ethereum node without
     setting up our own. > web3 = new Web3(new Web3.providers.
    HttpProvider("https://mainnet.infura.io/" + INFURA_API));
> web3.eth.blockNumber;
5289236
```

Web3py example:

```
$ ipython
In [1]: from web3 import Web3, HTTPProvider
In [2]: import os
In [3]: INFURA_API = os.environ['INFURA_API']
In [4]: w3 = Web3(HTTPProvider('https://ropsten.infura.io/'+
    INFURA_API))
In [5]: w3.eth.blockNumber
Out[5]: 2872088
```

B Scalability through Gas Saving masks

```
1  pragma solidity ^0.4.21;
2
3  contract Packing {
4
5      uint64  a;
6      uint64  b;
7      uint64  c;
8      uint64  d;
9      uint128 e;
10     uint128 f;
11
12     function set() public {
13         a = 1;
14         b = 2;
15         c = 3;
16         d = 4;
17         e = 5;
18         f = 6;
19     }
20 }
```

```
1 $ solc --optimize --asm Packing.sol | grep sstore | wc-l
2 2
3 $ solc --asm Packing.sol | grep sstore | wc -l
4 6
```

Figure B.1: Running the optimizer in storage variables less than 256 bytes results in 2 SSTORE commands instead of 6 which reults in significant savings in gas costs

Game interface:

```
pragma solidity ^0.4.21;

interface Game {
    event PlayerRegistered(uint16 playerID, address player);

    function Register() public returns (uint16 playerID);
    function CreateCharacter(uint256 creationTime, uint256 class,
       uint256 race, uint256 strength, uint256 agility, uint256
       wisdom,  uint256 metadata)   external;
    function GetCharacterStats(uint256 index) external view returns
        (uint16 playerID, uint32 creationTime, uint8 class , uint8
       race, uint16 strength, uint16 agility, uint16 wisdom,
       bytes18 metadata);

}
```

Figure B.2: Interface for described use-case

Tightly packed code:

```
struct Character {
    uint16 playerID;
    uint32 creationTime;
    uint8 class;
    uint8 race;
    uint16 strength;
    uint16 agility;
    uint16 wisdom;
    bytes18 metadata;
}
```

(a) Character structure definition

```
function CreateCharacter(
    uint256 creationTime,
    uint256 class,
    uint256 race,
    uint256 strength,
    uint256 agility,
    uint256 wisdom,
    uint256 metadata)
    external
{
    uint16 playerID = player2ID[msg.sender];
    require(playerID != 0);

    Character memory  c;
    // Overhead from converting, in order to match interface
    c.playerID = uint16(playerID);
    c.creationTime = uint32(creationTime);
    c.class = uint8(class);
    c.race = uint8(race);
    c.strength = uint16(strength);
    c.agility = uint16(agility);
    c.wisdom = uint16(wisdom);
    c.metadata = bytes18(metadata);

    uint CharacterId = Characters.length;
    emit CharacterCreated(c, CharacterId);

    Characters.push(c);
}
```

(b) Create character simply sets values to each struct variable

```
    function GetCharacterStats(uint256 index)
        external view
        returns (
            uint16 playerID,
            uint32 creationTime,
            uint8 class,
            uint8 race,
            uint16 strength,
            uint16 agility,
            uint16 wisdom,
            bytes18 metadata
        )
    {
        Character memory  c = Characters[index];
        return (
            c.playerID,
            c.creationTime,
            c.class,
            c.race,
            c.strength,
            c.agility,
            c.wisdom,
            c.metadata
       );
}
```

(c) Retrieve the character and save it in memory, then return all values.

Figure B.3: Implementation requires a Solidity 'struct' to pack all the variables together. CreateCharacter and GetCharacterStats

Method 2 code:

```
function CreateCharacter(
  uint256 creationTime,
  uint256 class,
  uint256 race,
  uint256 strength,
  uint256 agility,
  uint256 wisdom,
  uint256 metadata)
  external
  {
      uint16 playerID = player2ID[msg.sender];
      require(playerID != 0);

      uint c = uint256(playerID);
      c |= creationTime  <<  16;
      c |= class         <<  48;
      c |= race          <<  52;
      c |= strength      <<  56;
      c |= agility       <<  72;
      c |= wisdom        <<  88;
      c |= metadata      << 104;

      uint CharacterId = Characters.length;
      emit CharacterCreated(c, CharacterId);

      Characters.push(c);
  }
```

Figure B.4: Create Character by shifting variables

```
    function GetCharacterStats(uint256 index)
        external view
        returns (
            uint16 playerID,
            uint32 creationTime,
            uint8 race,
            uint8 class,
            uint16 strength,
            uint16 agility,
            uint16 wisdom,
            bytes18 metadata)
        {
        uint c = Characters[index];
        return (
            uint16(c),
            uint32(c >> 16),
            uint8((c >> 48) & uint256(2**4-1)),
            uint8((c >> 52) & uint256(2**4-1)),
            uint16(c >> 56),
            uint16(c >> 72),
            uint16(c>> 88),
            bytes18(c>>104)
           );
    }
```

Figure B.5: Get the traits of a character by shifting and masking appropriately. Typecasting is the same as applying a mask of N bits.

Method 3 code:

```
1     function GetProperty(bytes32 Character, uint mask, uint shift)
          private pure returns (uint property) {
2         property = mask&(uint(Character)/shift);
3     }
4
5     function SetProperty(bytes32 Character, uint mask, uint shift,
          uint value) private pure returns (bytes32 updated) {
6         updated = bytes32((~(mask*shift) & uint(Character)) | ((
              value & mask) * shift));
7     }
```

(a) Getting and setting a property

```
1     uint private constant mask32        = (1 << 32) - 1;
2     uint private constant _CreationTime = 1 << 16;
```

(b) Mask and shift offsets for CreationTime

```
1     function SetCreationTime(bytes32 Character, uint256 value)
          internal pure returns (bytes32) { return SetProperty(
          Character, mask32,  _CreationTime, value); }
2     function GetCreationTime(bytes32 Character) internal pure
          returns (uint32) { return uint32(GetProperty(Character,
          mask32, _CreationTime)); }
```

(c) Getting and Setting creation time API

Figure B.6: Parts of the Library API for Character Creation

```
1 bytes32 c = c.SetPlayerID(
      playerID);
2 c = c.SetCreationTime(
      creationTime);
3 c = c.SetClass(class);
4 c = c.SetRace(race);
5 c = c.SetStrength(strength);
6 c = c.SetAgility(agility);
7 c = c.SetWisdom(wisdom);
8 c = c.SetMetadata(metadata);
```

(a) Create Character by shifting variables

```
1  bytes32 c = Characters[index];
2  return (
3      c.GetPlayerID(),
4      c.GetCreationTime(),
5      c.GetClass(),
6      c.GetRace(),
7      c.GetStrength(),
8      c.GetAgility(),
9      c.GetWisdom(),
10     c.GetMetadata()
11 );
```

(b) get character variables

Figure B.7

C Security

```
1
2 $ ipython
3 In [1]: from web3 import Web3, HTTPProvider
4 In [2]: import os
5 In [3]: INFURA_API = os.environ['INFURA_API']
6 In [4]: w3 = Web3(HTTPProvider('https://mainnet.infura.io/'+
     INFURA_API))
7 In [5]: w3.eth.getStorageAt(w3.toChecksumAddress('0
     xd8993F49F372BB014fB088eaBec95cfDC795CBF6'),  0)
8 Out[5]: HexBytes('0
     x0000000000000000000000000000000000000000000000000000000000000001
     ')
```

Figure C.1: Inspecting the first storage slot of a contract

D Code for Smart Meters