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

Georgios Konstantopoulos

Contents

List of Figures

List of Tables

Abstract

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. 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. We apply the gained insight and techniques on the described metering and billing use case in order to enhance its viability and robustness in production. Finally, we evaluate the effectiveness of the proposed scalability techniques and security best-practices on the written smart contracts.

1 Introduction

1.1 Motivation

In 2009 Satoshi Nakamoto published the Bitcoin whitepaper [1] 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 on top of Bitcoin [2], in the end of 2014 a blockchain platform which was capable of executing smart contracts was released, called Ethereum [3]. A smart contract, first referred to as a term by Szabo in [4], is software that is executed on a blockchain and can be used as a framework for secure and trustless computation.

Smart contracts became very popular because they enabled the concept of Decentralized Applications (DApps). Use cases include crowdfunding through Initial Coin Offerings (ICO), decentralized exchanges where order matching and settling does not require a central operator, and decentralized autonomous organizations where token shareholders can vote on proposals. Smart contracts operate as code with the potential to hold large amounts of funds for a particular use case. Past events have shown that security is very important as large financial amounts have been stolen from smart contracts. Finally, due to the distributed nature of blockchains there are challenges towards achieving scalability and high transaction throughput, which traditional centralized payment processors or server architectures provide.

1.2 Research Questions

We aim to answer the following research questions in this thesis:

1. How can the scalability of smart contracts be improved?

2. How can the security of smart contracts be improved?

3. How can a system that is able to measure and bill the energy consumed by a set of energy meters be modeled?

The system must be able to perform accounting on the measured energy based on a pre-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.

1.3   Scope

This Master Thesis explores the fundamental terms needed to understand blockchain terminology. The contributions to scalability are limited to optimizing smart contracts. Other scalability solutions are mentioned but in depth analysis is considered out of scope. On security we limit ourselves to the industry's best practices, as per the literature and utilize a tool provided by an auditing firm. The architecture of the metering and billing model of energy consumed by the set of energy meters mentioned in Section 1.2 is implemented based on a specific use-case set by Honda Research and Development Germany, hereafter referred to as HREG, with which we collaborate for the purpose of the Master Thesis.

The technology used includes but is not limited to the Solidity programming language[1], the Truffle Framework for streamlining the compilation, testing and deployment process, Javascript for unit testing the smart contracts and the Python programming language for automating tasks, performing plotting and analyzing of data.

1.4   Outline

In Chapter 2 introduces terminology required for understanding blockchain fundamentals. We then proceed to explain how Ethereum works at a lower level, along with the programming techniques and tooling necessary to author smart contracts.

In Chapter 3 we refer to the scalability issues that plague today's blockchain systems and provide a brief description of the known possible solutions that can be implemented to solve these problems. We make a contribution on scaling smart contracts which allows for more optimized writes to the Ethereum blockchain.

In Chapter 4 we go over the most significant security issues found in Ethereum and its smart contracts. Having understood these, we evaluate and augment the auditing tool *Slither*'s ability to detect and identify vulnerabilities and compare it to taxonomy of tools described in [5]. Finally, we analyze smart contracts *honeypots* as a technique used by adversaries to profit from existing and well known smart contract flaws.

In Chapter 5 we present ways in which smart contracts can address the energy market's inefficiencies and create a suite of smart contracts which are able to answer research question 3 from 1.2, while taking the lessons learned from 3 and 4 into account.

In Chapter 6 we evaluate the performance and the extent at which the smart contracts from Chapter 5 achieve their goal.

In Chapter 7 we reiterate and summarize on the findings from the previous chapters, highlight future work that can be done to further improve both the design

---

[1]Ethereum's most popular language for writing smart contracts

of the described smart contracts but also the security and scalability of Ethereum.

1.5   Writing Conventions

A glossary can be found at the end of the document for all terminology used (written in italics). Limited code snippets can be found across the document when needed for explanatory reasons, full code with explanations can be found in the Appendices and in the accompanying GitHub URL. Minimal parts of Chapter 5 are redacted due to the inclusion of intellectual property of HREG.

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.

A blockchain can be described as a list of *blocks* that grows over time. Each block contains various metadata (called *block headers*) and a list of transactions. A block is linked to another block by referencing its *hash*. A blockchain gets formed when each existing block has a valid reference to the previous block. It is the case that as more blocks get added to a blockchain, older 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 specific to the implementations of the Ethereum Platform. The Ethereum Yellowpaper provides details on the formal definitions and contents of each term [6].

2.1.1 Cryptographic Hash Functions

A hash function is any function that maps data of arbitrary size to a fixed size string. The result of a hash function is often called the *hash* of its input. Hash functions used in cryptography must fullfil additional security properties and are called *cryptographic hash functions*.

More specifically, a secure cryptographic hash function should satisfy the following properties[1] [7]:

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 $H(x') = H(x)$. 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.

---

[1] $H(x)$ refers to the hash of x

Bitcoin uses the SHA-256 cryptographic hash function, while Ethereum uses KECCAK-256 [8, 9]. Ethereum's KECCAK-256 is oftentimes mistakenly referred to as SHA-3 which is inaccurate since SHA3-256 has different padding and thus different values[10].

2.1.2 Public Key Cryptography

Also referred to as Asymmetric Cryptography, it is a system that uses a pair of keys to encrypt and decrypt data. The two keys are called *public* and *private*[2]. 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[11] or calculating the discrete logarithm[3] for the Elliptic Curve Digital Signature Algorithm (ECDSA), being hard.

Public key cryptography allows for secure communications by achieving:

1. **Confidentiality - A message must not be readable by a third party.**

   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 - The receiver must be able to verify the sender's identity.**

   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.

3. **Confidentiality and Authenticaton - The receiver must be able to verify the sender's identity and verify that the message was not read by a third party.**

   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 the sender's public key.

4. **Integrity - The receiver must be able to verify that the message was not modified during transmission.**

   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:

---

[2]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.

[3]The discrete logarithm $log_b a$ is an integer k such that $b^k = a$

(a) The sender calculates the hash of the message that they are transmitting and concatenates the message with the hash.

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

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

(d) The receiver hashes the plaintext and compares the result to the transmitted hash.

(e) 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 would 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.

2.1.3   Basic features of blockchains

A blockchain acts as a distributed immutable public ledger. It can be used to efficiently transfer value between multiple parties without using a trusted intermediary (e.g. a bank in the case of a financial transaction) to settle the transaction.

Due to the public nature of the ledger, it provides transparency as every transaction can be inspected to verify its associated information. This can be utilized by interested suppliers (e.g. companies) to cultivate trust with their clients or partners by providing them access to the needed functionalities without compromising otherwise private information. There are privacy implications with this however, since companies might not want to disclose all pieces of information to the public. Privacy enabled blockchains which use advanced cryptography to hide transaction information from non-transacting parties already exist [12, 13, 14] . Bitcoin utilizes *pseudonyms* to hide the identity of an individual behind a random address[4], however research has shown that this is not always a reliable privacy preservation measure [15, 16].

Due to the distributed nature of the ledger, it is currently impossible for a party to censor a transaction. This has very powerful implications (e.g. in the case of an authoritarian regime, there is no way for a transaction to be cancelled due to a court order). On the other hand, in the case of theft of private keys, there is no way to prevent an attacker from stealing a victim's funds. At current scale, transactions on a blockchain have very low costs to execute and very high confirmation speeds [17].

The ledger is secured by the used protocol rules which utilize so called *consensus algorithms* to provide transaction finality and immutability. As a result, blockchains can be used to timestamp events in history [18] which can be utilized to prevent fraud. This feature is incompatible with the flexibility of centralized databases which allow for any past event to be rewritten, which allows for scenarios such as reverting the bank balance of a customer in the case of a mistaken transaction.

---

[4]An individual can generate multiple addresses from the same private key.

2.1.4 Blockchain Types

In order to tackle the disadvantages mentioned in 2.1.3, blockchains that make tradeoffs in decentralization for more 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 network. This is one of the main benefits of using a blockchain, transparency.

There are two blockchain categories:

1. **Public or Permissionless.**

   There is low barrier to entry, they have full transparency and immutability.

2. **Private or Permissioned.**

   Participation is limited based on the rules set by a group of operators. There are added privacy features and immutability is not absolute.

Vitalik Buterin goes indepth in the advantages and disadvantages between private and public blockchains in [19]. Due to the scalability and privacy restrictions of public blockchains, corporations can use permissioned blockchain frameworks [20, 21, 22] to include blockchain technology in their processes.

2.2 Ethereum Blockchain

The Ethereum blockchain acts as a transactional state machine. The first state is the *genesis* state referred to as the *genesis block*. After the execution of each transaction, the state changes. Transactions are collated together in *blocks*.

Figure 2.1: Ethereum can be seen as a chain of states [23]

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[5] and is able to validate their block proposal. This is a process known as *Proof of Work* (PoW) [24].

Due to a large number 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[6]. Each blockchain protocol has a way to resolve forks and determine which chain is the valid chain. In Ethereum the longest chain is based on total difficulty[7] which can be found in the *blockheader*. Ethereum is advertised to be using a modification of the GHOST Protocol [25] as its chain selection mechanism which uses *uncle blocks*[8]. This is contradictory since Ethereum's uncle blocks do not count towards difficulty and as a result, Ethereum does not actually use an adaptation of the GHOST protocol [26]; the uncle reward is just used to reduce miner centralization.
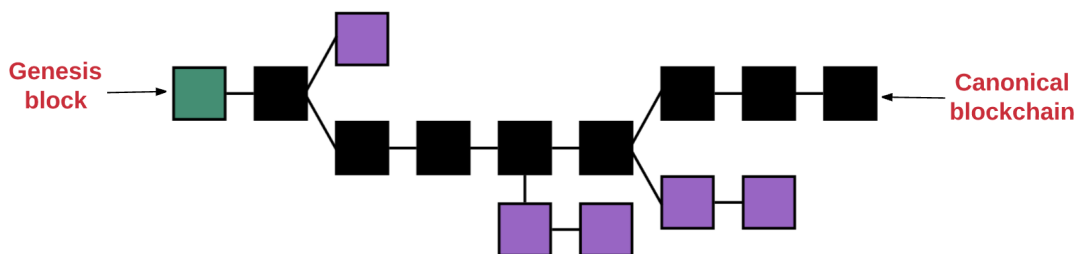
Figure 2.2: The Ethereum protocol chooses the longest chain [27]

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

World State

Ethereum's world state is a mapping between addresses of accounts and their states. Full nodes download the blockchain, execute and verify the full result of every trans-

---

[5]The Ethereum network's native currency

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

[7]Difficulty is a measure of how much computational effort needs to be given coon average by a mir to solve a PoW puzzle. Total Difficulty is the sum of the difficulties of all blocks until the examined block

[8]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

action since the genesis block. Users should run a full node if they need to execute every transaction in the blockchain or if they need to swiftly query historical data.
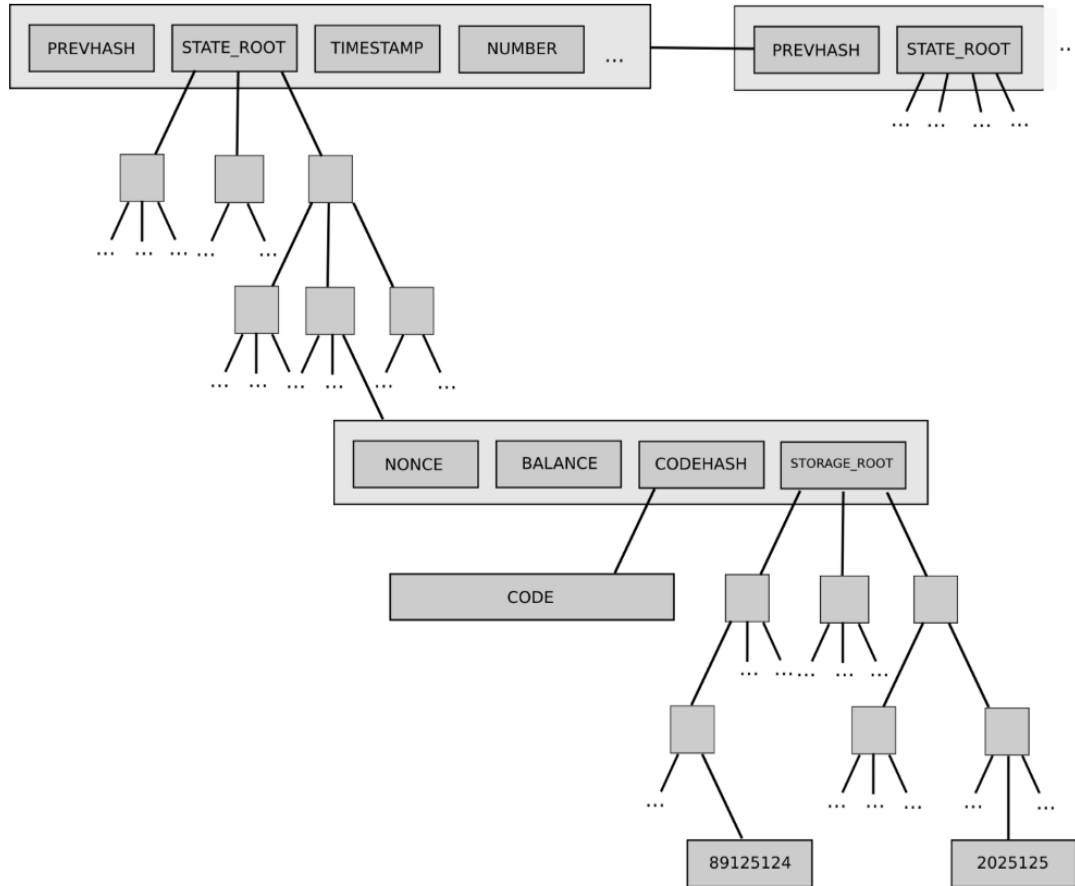
Figure 2.3: The Ethereum world state [3]

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 without storing the entire tree. A *Merkle Tree* is a binary tree where each parent node is the hash of its two child nodes[9].

---

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

Figure 2.4: Node calculation in a Merkle Tree [28]

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: Merkle path calculation [28]

2.3.1.1 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). EOAs are controlled by their Private Key and cannot contain EVM code. CAs contain EVM code and are controlled by the EVM code,

Figure 2.6: Externally Owned Accounts and Contract Accounts in the Ethereum World state [23]

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: Transaction made by an EOA to an EOA or a contract [27]

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' [6]. The address of a CA is deterministically computed during contract creation from the sender EOA account's address and their transaction count[10].

---

[10]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'[11]

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 a specially formatted data structure that gets signed by an EOA[12] and gets broadcasted 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[13].

4. **gas:** The maximum amount of gas units (hereafter referred to as *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.

---

[11]1 ether = 10^{18} wei

[12]With the EOAs private key

[13]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 of 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.

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, an attacker would be able to maliciously flood the network with computationally intense transactions and cause nodes to perform costly operations for extended periods of time. Ethereum uses gas to introduce a cost on performing computations. Gas manifests itself as the fees to be paid by the sender for a transaction to complete successfully.

Every computational step on Ethereum costs gas. The simplest transaction which involves transferring Ether from one account to another costs 21000 gas. Calling functions of a contract involves additional operations where the costs can be estimated through the costs described in [29, 6].

When referring 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:

$$totalTransactionCost = gasPrice * gasUsed \qquad (2.1)$$

where *gasUsed* is the amount of gas consumed by the transaction.

$$()$$

Miners are considered to be rational players who are looking to maximize their profit. As a result, they are expected to include transactions with transaction costs before transactions with low transaction costs. This effectively creates a 'fee market' where users are willing to pay more by increasing the *gasPrice* to have their transactions confirmed faster. In the times of network congestion such as popular Initial Coin Offerings[14][30] or mass-driven games such as CryptoKitties[15][31] transactions become very expensive and can take long times to confirm.

2.3.4.1 Successful Transaction

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

---

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

Figure 2.8: Successful transaction [27]

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 occurring during its execution. In this case, any gas consumed goes to the miners and any state changes that would happen are reverted. This is similar to the SQL transaction commit-rollback pattern.

Figure 2.9: Failed transaction that ran out of gas [27]

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. If there were no rules for block creation, a malicious node would be able to consistently censor transactions or double-spend [32]. In order to avoid that, consensus algorithms elect a network participant to decide on the contents of the next block.

Ethereum uses a consensus algorithm called ethash[9] which is a memory-hard[16] consensus algorithm which requires a valid PoW in order to append a block to the Ethereum blockchain. PoW involves finding an input called *nonce* to the algorithm

---

[16]Requires a large amount of memory to execute it. This means that creating ASICs for ethash is harder, although not impossible [33]

so that the output number is less than a certain threshold[17]. PoW algorithms are designed so that the best strategy to find a valid nonce is by enumerating through 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 the *block reward* 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 (PoS), which deprecates the concept of 'mining' and replaces it with 'staking'. PoS is considered to be a catalyst for achieving scalability in blockchains and is briefly discussed in Chapter 3.

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 by the EVM.

2.4.1   Programming Languages

Languages that compile to EVM code are Solidity, Serpent, LLL or Vyper. Solidity is the most popular language in the ecosystem and although often comparable to Javascript, we argue that Solidity 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 the compiler.

```solidity
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

Due to the nascence of these languages and the security mistakes that have occurred due to them providing programmers with powerful state-changing functions, active research is being done towards safer languages [34].

---

[17]The threshold is also called difficulty and adjusts dynamically so that a valid PoW is found approximately every 12.5 seconds

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 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 flaw is found on a deployed contract, the only way to fix it is by deploying 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 [35] consensus algorithm, while Ropsten is running Ethash [9] with less difficulty.

We provide a 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. 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[18](User Interface at A.3, 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[19] 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[20]. We illustrate an example of connecting and fetching the latest block from Ropsten and Mainnet using Web3.js in A.4 and Web3.py in A.5. The full specifications of each library's API can be found in their respective documentation[21][22]

---

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

2.4.2.3   Truffle Framework

The Truffle Framework is a development framework for smart contract development written in NodeJS. It is currently the industry standard for developers. It allows for automating the smart contract deployment pipeline through *migration* scripts and scripting test suites for scenarios using the Mocha Testing suite. Finally, it includes a debugger for stepping through transaction execution and can internally launch a ganache testnet.

2.5   Related Work

Techniques which illustrate more efficient smart contracts by storing less data on a blockchain are described in [36]. Cheng at al makes it clear that compiler optimizations in smart contracts still need improvements in order to avoid unnecessary expenses [37]. On network level scalability there are various approaches such as executing transactions 'off-chain'[23] and use a blockchain only for the final settling of a series of transactions [38, 39, 40]. Other approaches exist which involve creating 'sidechains' which can be used to offload the computational effort from the 'mainchain' [41, 42, 43, 44, 45]. Finally, another approach to achieving scalability is via permissioned blockchains which trade decentralization and transparency for efficiency [21, 46].

Extensive analyses have been performed on the security of blockchains as networks [26, 47] and specifically on the security of Ethereum smart contracts [48] which have proven to be insufficiently secure for the amounts of funds that they hold. As a result, tools that are able to analyze both source code and compiled bytecode for vulnerabilities have been developed [49, 50, 51, 52, 53, 54]. A recent study [55] illustrates how smart contracts that can freeze or cause loss of funds can be detected [56].

Utilizing blockchain for Internet of Things is explored in [57, 58], while a model for billing and accounting with smart contracts is proposed in [59]. Energy market use-cases are being piloted by [60, 61] and prototypes are being tested such as [62, 63].

---

[23]An exchange of cryptographically signed messages that does not happen on a blockchain.

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.With Ethereum's current blockGasLimit at 8003916, if all transactions in Ethereum were simple financial transactions[1], each block would be able to verify 381 transactions, or 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 because 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 inaccurate. 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. In the Ethereum Github, this is described as the *Scalability Trilemma*.

> **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)

---

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

- 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)

(Scalability Trilemma, Sharding FAQ)

An example that trades decentralization for more transactions is increasing the block size so that more transactions can fit inside a block and thus increase throughput. 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 [64], Proof of Work is a consensus algorithm optimized for censorship-resistance while (in theory) maintaining a low barrier to entry, as shown in Figure 3.1. Bitcoin and Ethereum's PoW networks have slow probabilistic time to finality and do not scale well. However, due to economies of scale, PoW blockchains end up being centralized around small numbers of miners [65].
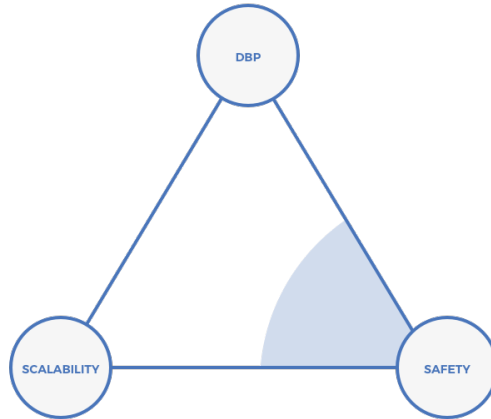
Figure 3.1: The Scalability Triangle [64]

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

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

---

[2]Lock up cryptocurrency for an amount of time and get paid in interest after a predefined time period.

is still under heavy research. The Ethereum Foundation is working on 'Casper the Friendly Finality Gadget' [66] 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] [67] will enable a full transition to Proof of Stake.

Sidechains

A sidechain [41] is a blockchain defined by a custom 'rule-set' whihh cincbe 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' [45] and involves creating *Plasma chains* that run their own consensus algorithm and communicate with the mainchain via a two-way peg as described in [41]. *Plasma chains* can have more adjustable parameters such as be less decentralized, however the protocol does not allow for the Plasma Chain operator to abuse their power. A more recent Plasma construct is called 'Plasma-Cash' [44] and describes a more efficient way of executing fraud proofs, in the case of a malicious actor in a *Plasma chain*.

Sharding

Due to the architecture of the EVM all transactions are executed sequentially on all nodes. Sharding [68] 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.

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].

3.3   Contract Level Scalability

In a recent study [37], after evaluating 4240 smart contracts, it is found that over 70% of them cost more gas than they should due to the compiler failing to properly optimize the Solidity code during compilation. In this section we explore how

---

[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 the channel with one more transaction at the end.

gas gets computed in smart contracts and potential ways we can save on gas and transaction costs.

3.3.1   Gas Costs

An Ethereum transaction total gas costs are split in two:

1. **Base 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 for a transaction (21000 gas).

   (b) Extra cost in the case that the transaction involves deploying a contract (32000 gas).

   (c) The cost for every zero byte of data in the transaction input field (4 gas per zero byte).

   (d) The cost of every non-zero byte of data in the transaction input field (68 gas per non-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 [6, 29].

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. Transactions that cost less gas allow more room for other transactions to be included in a block which can improve scalability.

Table 3.1: Gas costs for different operations [37]

Operation	Gas	Description
ADD/SUB	3	
MUL/DIV	5	Arithmetic operation
ADDMOD/MULMOD	8	
AND/OR/XOR	3	Bitwise logic operation
LT/GT/SLT/SGT/EQ	3	Comparison operation
POP	2	
PUSH/DUP/SWAP	3	Stack operation
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional jump
JUMPI	10	Conditional jump
SLOAD	200	
SSTORE	5,000/20,000	Storage operation
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[8]. The focus of this section will be to explore ways to decrease 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 [69] describes a method of buying gas at low cost periods and saving it in order to spend it when gas prices are

---

[7]Used to create a new contract.
[8]Used to store data operations

higher[9]. 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 this has lead to unexpected scenarios [70]).

2. Reuse code through libraries [71].

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. When iterating through an array, if the break condition involves the array's length set it as a stack variable the loop. This way, it doesn't get loaded during each loop and allows for saving 200 gas per iteration [37].

5. 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.

6. 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[10].

As described in [37] there is a lot of room for further compiler optimizations. Future Solidity compiler versions are addressing some already[11][12][13].

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. Refer to B.5 for an example of the optimizer's functionality.

3.3.2 Gas Savings Case Study

We proceed to compare the gas efficiency of 3 methods for storing data in a smart contract based on a gaming use-case. The contract design requirements are:

- A user must be able to register as a player in the contract.

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

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

---

[9]When the network is congested

[10]This pattern has been used in combination with IPFS, https://ipfs.io

[11]https://github.com/ethereum/solidity/issues/3760

[12]https://github.com/ethereum/solidity/issues/3716

[13]https://github.com/ethereum/solidity/issues/3691

Table 3.2: Required variables and size to describe a Character.

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

The choice of variables in Table 3.2 is made to represent what the traits of a character would be in a game built on a smart contract. 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.4.

We will examine the gas costs for deployment and for calling each function for the following implementations:

1. Packing of traits by utilizing Solidity's optimizer and `struct` variables

2. Manually pack traits in a `uint256` variable with masking and shifting.

3. Manually pack traits in a `bytes32` variable (equivalent in length to `uint256`) with masking and shifting, utilizing Solidity Libraries, influenced by [72].

We use `bytes32` in Method 3 because the bit operations done in Solidity when extracted in functions do not function as expected with `uint256` variables. The full contract implementations for each method can be found in Appendix B. A comparison of the performance for each method is shown in 3.3.3

3.3.2.1 Packing of traits by utilizing Solidity's optimizer and `struct` variables

We use Solidity's built-in `struct`[14] keyword as 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` can be accessed by its name as seen in B.5c, like the property of an object.

```
1  Character memory  c;
2  c.playerID = uint16(playerID);
3  c.creationTime = uint32(creationTime);
4  c.class = uint8(class);
5  c.race = uint8(race);
6  c.strength = uint16(strength);
7  c.agility = uint16(agility);
8  c.wisdom = uint16(wisdom);
9  c.metadata = bytes18(metadata);
```

```
1  Character memory  c = Characters[index];
2  return (
3      c.playerID,
4      c.creationTime,
5      c.class,
6      c.race,
7      c.strength,
8      c.agility,
9      c.wisdom,
10     c.metadata
11 );
```

(a) Method 1: `CreateCharacter`   (b) Method 1: `GetCharacterStats`

Figure 3.2: Method 1 API

---

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

In this case, assignment and retrieval of the variables is done in a very straight-forward 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.5c. The gas costs per function call with this method are shown in 3.3.

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

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

3.3.2.2 Manually pack traits in a uint256 variable with masking and shifting

In 3.3.2.1 we rely on the optimizer to make storing a character's traits more efficient. It turns out[15] that the optimizer is not able to remove all unnecessary operations and there is still room for improvement. In order to get better results, we create a local stack variable[16] which is large enough to store all the traits from 3.2. Instead of creating a `struct`, we manually encode each trait in the said variable, essentially we act as the optimizer, which results in much less gas spent as both the contract's bytecode is smaller and the `CreateCharacter` function is more efficient. We proceed to describe the encoding process.

```
1 uint c = uint256(playerID);
2 c |= creationTime  << 16;
3 c |= class          << 48;
4 c |= race           << 52;
5 c |= strength       << 56;
6 c |= agility        << 72;
7 c |= wisdom         << 88;
8 c |= metadata       << 104;
```

(a) Method 2: `CreateCharacter`

```
1 return (
2     uint16(c),
3     uint32(c >> 16),
4     uint8((c >> 48) & uint256(2**4-1)),
5     uint8((c >> 52) & uint256(2**4-1)),
6     uint16(c >> 56),
7     uint16(c >> 72),
8     uint16(c>> 88),
9   bytes18(c>>104)
10 );
```

(b) Method 2: `GetCharacterStats`

Figure 3.3: Method 2 API

Setting data requires shifting left N times and performing bitwise OR with the target variable, where N is the sum of the number of bits of all variables to the right of the target variable, as shown in Figure 3.4. This is implemented in B.6

---

[15]https://github.com/figs999/Ethereum/blob/master/Solc.aComedyInOneAct
[16]The data is 248 bits long, so we create a `uint256` variable

Figure 3.4: Encoding data in a `uint256` variable

Retrieving data requires shifting right N times and performing bitwise AND with the target variable's size, where N is the sum of the number of bits of all variables to the right of the target variable. Figure 3.5 illustrates retrieving the `creationTime` trait from the `uint256` by shifting right 16 times and performing bitwise AND with $2^{32} - 1$ since `creationTime` is a 32 bit variable. This is implemented in B.7.

Figure 3.5: Retrieving `creationTime` from the encoded `uint256`

The gas costs per function call with this method are:

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

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

3.3.2.3 Manually pack traits in a bytes32 variable with masking and shifting, utilizing Solidity Libraries

Code reusability and readability should always be given high priority. Although data packing is very efficient in 3.3.2.2 compared to 3.3.2.1, the code is hardly readable and it is impossible to reuse parts of it. We utilize Solidity's `Library` built-in which allows us to define a set of methods which can be applied on a datatype using the `using X for Y` syntax[17]. A simple example is shown in B.1.

The visibility of a Library's exported functions can be:

1. **Internal:** In this case, the compiled library's bytecode is inlined to the main contract's code. This results in larger bytecode during deployment, however each of the Library's functions are called via the `JUMP` opcode. In this case, only the base contract needs to be deployed.

---

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

2. **Public:** In this case, the main contract's bytecode has placeholder slots. These slots get filled by the Library's address which is obtained after deploying the Library contract. After replacing the placeholder slots with the Library's address, any function call to the library is done via the `DELEGATECALL` opcode.

Libraries with public functions are deployed as standalone contracts to be used by contracts made by other developers. This process is further described in B.8. They often include generic functionality such as math operations[18]. Depending on the complexity of the contracts, this can be more efficient compared to using `internal` functions.

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 store and retrieve a character's traits in a user-friendly manner.

```
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) Method 3: `CreateCharacter`

```
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) Method 3: `GetCharacterStats`

Figure 3.6: Method 3 API

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 contract deployment costs and the extra costs incurred when making function calls. We use `internal` because it requires less gas and since this is a specialized use-case it is not expected to be used by third-parties.

The gas costs per function call with this method are:

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

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

3.3.3 Results

Observing 3.3, 3.4 and 3.5, it is seen that in all cases the optimizer's first iteration creates significant gas savings. Further optimizer runs result in more gas expenditure during deployment but less per function call. This happens because `solc` optimizes

---

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

either for size or for runtime costs[73]. In this section we compare the gas costs for calling the `CreateCharacter` function and for deploying the contract for 1 optimizer run[19].
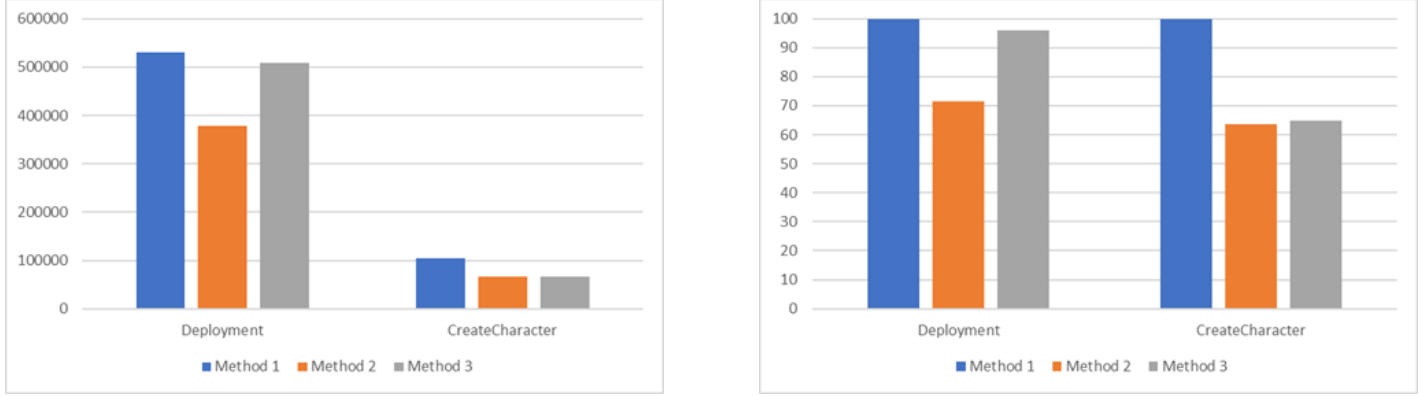
Figure 3.7: Gas cost comparison between the 3 proposed methods.

It is clear that the biggest savings are achieved through Method 2. Deploying a contract with Method 2 is 29% more efficient than Method 1, while Method 3 is only 4% more efficient than Method 1. Calling `CreateCharacter` is approximately 36% more efficient in both methods compared to Method 1.

Method 2 is the most efficient in terms of gas, however the coding style used for it is extremely compact and non-verbose, which makes difficult to maintain and modify the code, in case software requirements change, as seen in 3.3. In addition, there is no measure taken for overflows, so in case the `CreateCharacter` function gets called with an argument that is larger than the designed size, the result is miscalculated (B.2)

In an attempt to improve the readability and maintainability of the method by extracting the bit shifting logic to functions the deployment gains over Method 3 are in the range of 5%, and the function costs differ in the range of 0.001%, thus negating most of the gas efficiency advantages.

On the other hand, by comparing Method 1 (Figure 3.2) with Method 3 (Figure 3.6) it is seen that they have very similar syntax, with Method 1 being much more efficient in terms of gas. In addition, the bit shifting logic is extracted to a library which makes the code fully portable. Finally, there is more flexibility when performing bit operations due to the usage of `bytes32` in the library and as a result there is no risk of variables overflowing, contrary to Method 2.

In order to effectively compare and evaluate which method is the most effective for the described use case, we evaluate on a weighted score based on *Readability*, *Portability*, *Gas Efficiency* and *Security*, as illustrated in Table 3.6.

---

[19]We do not compare `Register` because the implementation is the same across all 3 methods and `GetCharacterStats` does not cost gas to call externally because it is a `view` method which does not modify the state.

Table 3.6: Selected weights for evaluation criteria.

Name	Weight	Description
Readability	0.25	How readable is the codes
Portability	0.2	How easy is it to use the code in another smart contract?
Gas Efficiency	0.3	How gas efficient is the code?
Security	0.25	How robust and secure is the code?

Taking the above into consideration, we choose to use Method 3 in order to make the implementation described in Chapter 5 more gas efficient while retaining its readability and maintainability. The results of the evaluation are shown in Table 3.7:

Table 3.7: Evaluation of each method

Name	Weight	M1	M1 (W)	M2	M2 (W)	M3	M3 (W)
Readability/Maintainability	0.25	5	1.25	3	0.75	4	1
Portability	0.2	5	0.8	3	0.6	4	0.8
Gas Efficiency	0.3	1	0.3	5	1.5	4	1.2
Security	0.25	5	1.25	4	1	4	1
Total	1	16	3.8	15	3.85	16	4

4 Ethereum and Security

The Ethereum platform itself has proven to be robust and reliable 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 flood 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 [74]. As a response, two hard-forks[1] were proposed [75, 76]. 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   [47] 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 Bitcoin which 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 Ethereum as a network, 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, the 'TheDAO' and the 'Parity Multisig'[5] (two independent incidents).

4.1.2.1 TheDAO

TheDAO is an acronym for 'The Decentralized Autonomous Organization'. The goal of TheDAO was to create a decentralized business fund where token holders would vote on projects worthy of being 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 the `splitDAO` function in the smart contract. The function was vulnerable to a *reentrancy*[6] attack which allowed an attacker to make unlimited withdrawals from the contract[77, 78].

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 (12% of Ether in existence at the time). 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' [79].

4.1.2.2 Parity Multisig 1

In July 2017 a vulnerability was found in the Parity Multisig Wallet which allowed an attacker to steal over 150.000 ether [80]. 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 [81] 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 involved the Library contract's `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,

---

[5]A multisig is a multisignature 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. The Parity Multisig refers to a multisig wallet implementation by Parity Technologies, `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 consecutive withdrawals and withdraw bigger amounts than their balance allowed.

however due to it being unprotected it was callable by any user of the wallet[7]. As a result, a malicious user could reinitialize any multisig with their address 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 cause of this hack.

4.1.2.3   Parity Multisig 2

After the first Parity hack, a new multisig wallet library was deployed, with access control in the `initWallet` function. This provided the expected functionality to all Parity Multisig implementations which were using the library. The fix in `initWallet` involved adding a `only_uninitialized` modifier which would only allow modification of the linked multisignature wallet owners during initialization. However, the `initWallet` function was never called on the library contract itself. 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[8] first became owner of the library by calling `initWallet` and then proceeded to delete the library by calling the `kill` function. This resulted in **all** contracts that were using the library's logic to be rendered useless, effectively *freezing* 513774 Ether, as well as tokens [82].

A number of proposals were made [83] in order to recover the locked funds. All of these would require an 'irregular state change' similar to what happened with the DAO[9] 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. Companies with public smart contracts also engage in bug-bounties, in which they reward users that responsibly disclose vulnerabilities found in their smart contracts. Comprehensive studies on identifying the security, privacy and scalability of smart contracts [85] as well as taxonomies aiming to organize past smart contract vulnerabilities have been done [48, 5], however due to the rapid evolution of the field they get outdated very soon.

In order to improve the security of deployed smart contracts, developers must learn to use automated auditing tools and also use the latest version of basic tooling like the Solidity Compiler. As an example, none of the tools mentioned in [5]

---

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

[8]They claim they were not acting as an adversary.

[9] 12 million ETH were moved from the "Dark DAO" and "Whitehat DAO" contracts into the WithdrawDAO recovery contract[84]

were able to detect the 'Uninitialized Storage Pointer' vulnerability[10], however the Solidity Compiler was later updated to throw a Warning if this issue exists.

4.2.1 Automated Tools

Auditing smart contracts is 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 to audit smart contracts. We process all the smart contracts with the latest version of the Solidity compiler, v0.4.22, in order to obtain have access to the latest warnings and errors.

As Slither is a static analyzer and works on the source code, its modules (called 'detectors') are able 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', but it is not able to perform symbolic execution and go through all the possible states of a smart contract like Oyente [49] or Mythril [50]. We describe the currently supported detectors:

Constant/View functions that write to state: Detects `view` functions that modify a smart contract's state. A `view` function uses the `STATICCALL` opcode [86] and as a result is unable to modify the state of a smart contract. The latest version of Solidity does not enforce this.

Misnamed constructors that allow modification of 'owner'-like variables: A constructor in a smart contract is a function that is named after the contract name and is run once at deployment. It initializes the contract state and usually sets an `owner` variable which allows the contract owner to have extra administrative permissions on the contract. If a function does not have the same name as the contract then it is callable at any time after deployment. In past cases, constructors were not named properly and were callable by adversaries who would claim ownership of the contract, leading to loss of funds [48].

Reentrancy bugs: Due to the TheDAO incident (4.1.2.1) a lot of attention was raised on reentrancy and race-to-empty[11] issues and as a result detecting this class of vulnerability is standard in all auditing tools.

Deleting a struct with a mapping inside: Calling `delete` on a `struct` object with a `mapping` variable does not clear the contents of the mapping[12]. This has not been exploited in the wild, however it can be critical in the case of a banking DApp that keeps tracks of balances. A full Proof of Concept is given in C.0.1

Variable Shadowing: When a contract (child) that inherits from another contract (parent) defines a variable that already exists in the parent, two separate instances of the variable get stored in the deployed contract. As a result, the parent's (child's) functions can modify only the parent's (child's) state variables. This is currently undetectable by other automated security tools. We further describe how this can get exploited in Section **??**.

Unprotected Function Detection: If the visibility modifier of a function is

---

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

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

[12]This is warned for in `http://solidity.readthedocs.io/en/v0.4.21/types.html`

set to `public` or `external` without any access control mechanism, that function is considered unprotected. This was the cause of the first Parity Wallet hack.

In addition to detecting potential vulnerabilities in smart contracts, Slither's detectors can be used to identify potential code styling issues and recommend fixes:

Similar Naming between Variables: Warns users in the case two variables with same length have very similar names. This is added to encourage more clear and verbose variable naming.

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

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

Wrong Event Prefix: As per the best practices, the names of 'events' should be capitalized. After a discussion on Github[13], using 'emit' for events is going to be mandatory for Solidity 0.5.0 and onwards.

Slither can be used both for finding known vulnerabilities, but also to avoid common anti-patterns. Due to its modular API we can extend it to include more rules. We contributed to the Slither repository by adding support for detecting usage of `tx.origin` and `block.blockhash`. The usage of `tx.origin` should be avoided unless necessary, and as stated in the Solidity documentation can incur in loss of funds[14]. `block.blockhash` has been misused in smart contracts and has resulted in amounts up to 400 ETH to be stolen [87]. We also contributed to the improvement of the accuracy of the module `UnimplementedFunctionDetection`. Finally, we implemented a detector to find unassigned calls to a library's functions (calling a library function on a variable does not assign its result back to the variable, it needs to be assigned manually). Table 4.1 is a modified version of Table 12 from [5], including Slither after our contributions, excluding issues that are now detectable by the Solidity compiler.

Table 4.1: Slither is able to detect more vulnerabilities compared to other tools.

Security Tool	Re Entrancy	TOD	High Gas	Unprot. Function	No Constructor	tx.origin	blockhash	Var. Shadowing	Mapping in struct
Oyente	Y	Y	N	?	?	N	N	N	N
Remix	Y	Y	Y	?	?	Y	Y	N	N
Securify	Y	Y	N	?	?	Y	N	N	N
SmartCheck	Y	Y	Y	?	?	Y	N	N	N
Mythril	Y	Y	N	?	?	Y	N	N	N
Slither	Y	N	N	Y	Y	Y	Y	Y	Y

Since the second Parity bug 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. Hackers who attempt to exploit them need to first deposit some amount before trying to drain the contract. Each honeypot has a mechanism to prevent the attacker from draining any funds. As a result, only the contract owner is able to withdraw the balance of the contract, including any stolen amounts. We proceed to describe the functionality of a honeypot which takes advantage of 'Variable Shadowing' in Solidity.

---

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

The contract in Figure 4.1 was deployed with an initial amount of 0.1 ether as a balance. At first, it seems that the `owner` variable can be manipulated by an adversary, by sending ether to the contract with an amount larger than the value of `jackpot`. After that, the adversary should be able to call `takeAll` and withdraw the funds from the contract.

```solidity
pragma solidity ^0.4.11;

contract Owned {
    address owner;    function Owned() {
        owner = msg.sender;
    }
    modifier onlyOwner{
        if (msg.sender != owner)
            revert();        _;
    }
}

contract KingOfTheHill is Owned {
    address public owner;
    uint public jackpot;
    uint public withdrawDelay;

    function() public payable {
        // transfer contract ownership if player pay more than
            current jackpot
        if (msg.value > jackpot) {
            owner = msg.sender;
            withdrawDelay = block.timestamp + 5 days;
        }
        jackpot+=msg.value;
    }

    function takeAll() public onlyOwner {
        require(block.timestamp >= withdrawDelay);
        msg.sender.transfer(this.balance);
        jackpot=0;
    }
}
```

Figure 4.1: A honeypot which takes advantage of Variable Shadowing.

TxHash	Block	Age	From		To	Value	[TxFee]
0x11713fad6712b7f...	4755628	116 days 19 hrs ago	0x00a0d727fe2a08e...	IN	0x4dc76cfc65b14b3...	0.42 Ether	0.00100878945
❗ 0xbbe67e38e60611...	4755591	116 days 19 hrs ago	0x00a0d727fe2a08e...	IN	0x4dc76cfc65b14b3...	0.42 Ether	0.000759255
0x577e17262ff3c2e...	4749448	117 days 20 hrs ago	0x22dc0138701299...	IN	0x4dc76cfc65b14b3...	0.21 Ether	0.00077133
0xfd5f91b4c34e232...	4527013	154 days 20 hrs ago	0x125d657d5cd16bf...	IN	0x4dc76cfc65b14b3...	0.100000000001 Ether	0.00003673
0x530a8ba6a4e7f97...	4525157	155 days 4 hrs ago	0x80028f80c7d5959...	IN	0x4dc76cfc65b14b3...	0.1 Ether	0.00008173
0x50f803e0bf9e782...	4525139	155 days 4 hrs ago	0x80028f80c7d5959...	IN	Contract Creation	0 Ether	0.000217967

Figure 4.2: Transactions trying to get ownership of the honeypot contract

Users tried to exploit it by depositing funds and expecting the ownership to

change to their address so that they would be able to withdraw the contract's funds. Contrary to what is expected, after sending enough ether to the contract, the `owner` variable which gets changed belongs to `KingOfTheHill` contract. The `takeAll` function, is decorated by the `onlyOwner` modifier[15] which looks for the value of `owner` in the `Owned` contract. As a result, the 'real' `owner` of the contract is unchanged, and the attacker is unable to claim ownership, effectively losing their funds. Running Slither on the contract can reveal the variable shadowing as shown in 4.3.

Figure 4.3: Slither is able to detect the variable shadowing in `owner`

4.2.2 Towards more secure smart contracts

Trustless smart contracts are impossible to modify after deployment. If a vulnerability is found, it is impossible to patch it. Developers 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 (e.g. an `ERC20` token contract) ensures that these parts of the code are going to be secured. Developers must be familiar with the security best practices as described by the industry's most sophisticated auditors[16], and should look for confluence between the results of different automated analyzers in order to filter out false-positives and find false-negatives. They should also be testing and improving their skills on platforms that have been set up with intentionally vulnerable contracts [88, 89, 90]. Finally, if compromises in the trustless nature of the contracts can be made, they can be made to be upgradeable, however that increases code complexity, gas costs and potentially introduces new attack vectors.

---

[15]A modifier is a special function which gets executed before or after the function it *modifies*. They are primarily used to enforce access control in functions.

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

5 Metering and Billing of Energy on Ethereum

Smart contracts can transform the energy industry. In this chapter we explore the inefficiencies of the energy market and identify gaps which can be filled by blockchain. We go through the advantages of an energy-based application built on smart contracts. Finally, we describe the business logic of a specific energy use-case which we implemented on Ethereum. The implementation takes into account the methods and concepts described in Chapter 3 and Chapter 4 in order to ensure that the smart contracts are efficient and robust.

5.1 Energy Market inefficiencies

The global energy market is gradually transitioning to clean and renewable energy. Regulations encourage usage of distributed energy resources (DERs) [91]. With the integration of DERs, consumers are gradually transforming to *prosumers* who store their energy surplus or sell it to their peers, effectively distributing the generation of energy, contrary to existing power systems which were designed to accommodate central points of energy generation.

There are numerous inefficiencies which need to be resolved in today's energy markets [92]:

1. **Transaction complexity:** More participants in the networks result in more complex transactions.

2. **Predictability and Reliability:** Availability of DERs is less predictable compared to traditional energy resources like coal.

3. **Empowered prosumer:** There need be monitoring tools and infrastructure that allow prosumers to manage their energy production and consumption.

4. **Geographic mismatches:** Locations that fully utilize DERs are usually far from key points of energy demand. Transmitting energy over long distances is inefficient.

5. **Trust and Security:** New participants will only choose the system if it is able to be trusted and is properly secured.

5.2 Energy Market use-cases for Blockchain

A number of use-cases for smart contracts have been identified in the energy market. We proceed to describe applications that derive from the advantages of blockchain (transparency, trustlessness, security) as discussed in Section 2.1.3.

1. **Supply chain tracking and optimization:** This is a broad blockchain use case which allows for optimization on logistics and tracking the location of products, reducing fraud and ensuring the validity of an event.

2. **Energy metering and billing:** By committing the values of energy consumed by entities to a blockchain, a timestamped record of meter readings gets generated which allows for the verification and more clear understanding of energy consumption across resources. This can be augmented to provide billing functionality (further described in 5.3).

3. **Peer to Peer (P2P) energy trading:** Consumers and prosumers operate in a local microgrid and trade energy between them, without a centralized operator, resulting in reduced loss of energy during transportation.

4. **P2P Energy Micropayments:** A car that is far from its charging station is charged by the surplus energy of another car. Since this is a short-lived process that happens often, such as during the waiting time at a traffic light, in order to send and receive micropayments between the involved parties, a blockchain can be used to utilize fast payments and low transaction fees.

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

We proceed to discuss the business logic of the use-case and then implement it. We utilize the Method 3 from Section 3.3.2.3 to optimize our smart contracts for gas efficiency and utilize Slither from Section 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, the contracts were tested and deployed on a private local blockchain, however it is possible to deploy the implementation to the Ethereum mainnet.

5.3.1 Metering of Energy

Smart Meters A smart meter is an energy meter which sends its measured energy readings to a monitoring server for logging. A smart contract must track each smart meter by a unique identifier and store each meter's power readings, along with the time of measurement. Due to the meter not being able to *ping*[1] its reading directly to the blockchain, we create a process where each the reading and timestamp of each meter is pulled from the monitoring server and then pushed to the blockchain.

---

[1]Hereafter, refers to a kWh,timestamp tuple which gets stored in the blockchain.
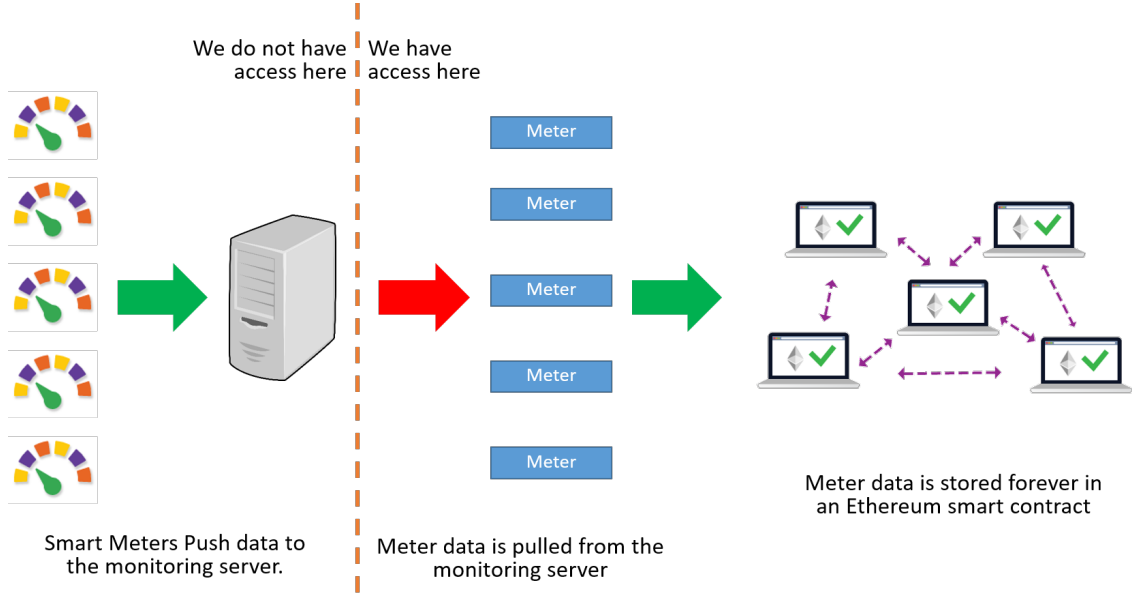
Figure 5.1: Meter Data gets pushed to the blockchain after getting pulled from the monitoring server

Virtual Meters   Smart meters get grouped in *Virtual Meters*. The purpose of a Virtual Meter (VM) is to track the consumption of multiple Smart Meters across various rooms of a company building. As a result, the power consumption of each VM is a function of the power consumptions of its linked smart meters.

The formula that describes each Virtual Meter is supplied from an internal partner and is considered to be known. An example formula can be seen in Equation 5.1. Variables prefixed with VM represent Virtual Meters, ELT and KMZ represent Smart Meters and F represent constants.

$$
\begin{aligned}
VM15 = \\
(ELT46 - (ELT31 + ELT16 + ELT17 + ELT35 + ELT36)) * F57 \\
+(ELT35 + ELT36 - ELT18) * F27 * F66 \\
+(KMZ6 - KMZ5) * (ELT16 + ELT17 + ELT18 + ELT19 + ELT20) \\
/KMZ2 * F27 * F66
\end{aligned}
\tag{5.1}
$$

It is the case that this process could be done entirely on-chain, by creating a smart contract that manages virtual meters separately. This would require another smart contract for virtual meters, which would need to store the Meter IDs for each meter associated to the virtual meter, along with the formula that describes how the virtual meter consumption is calculated. This logic is unnecessarily complex to be performed on a smart contract directly. Instead, it is done off-chain on the client side, by pulling the readings of each smart meter, calculating the reading of the virtual meter, and then pushing the aggregated reading to the smart contract's storage.
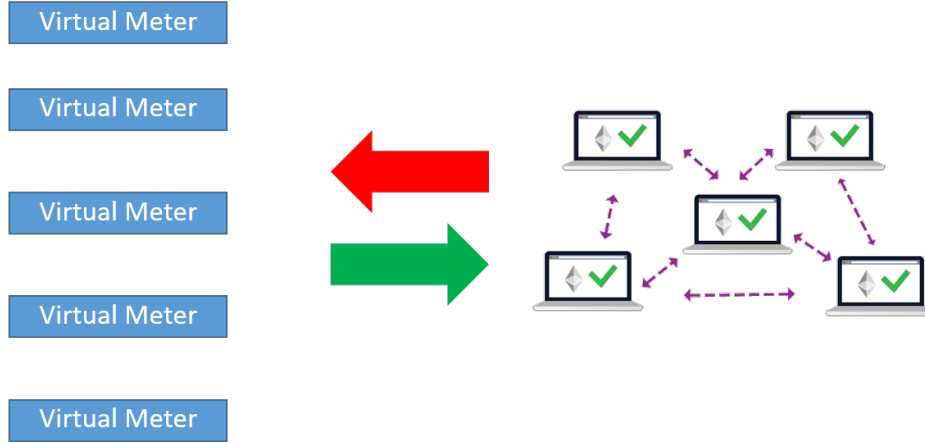
1. Meter data for each meter belonging to each VM gets pulled from the blockchain.
2. The energy consumption of each VM gets calculated, based on the formulas
3. Virtual Meter Readings get pushed back to the blockchain.

Figure 5.2: The consumption of Virtual Meters gets calculated from the consumption of their associated Smart Meters.

5.3.2 Accounting Logic

Business Units and *Departments* are terms used to describe the hierarchy for the accounting business logic of the company. Business Units are used for external accounting, while departments and their subdepartments are used for internal monitoring. In implementation terms, a Business Unit is equivalent to a Department.

A Department is composed of Virtual Meters, other Departments and Delegates. Virtual Meters get grouped in Departments. A Department can include other subdepartments of a lower tier. Finally, a Department may forward its consumption to other Departments called Delegates. A department's consumption is the sum of energy consumed by its Virtual Meters, its Departments and its Delegates.

Higher tier departments have the `headcount` property which defines the number of staff included in that Department. Delegation can be either through a headcount split or a fixed split. In headcount split, the amount that gets sent to each Delegate of the Department is proportional to their headcount percentage. Fixed split can only involve two Delegates, the first one receiving $X\%$ and the second one receiving $(100 - X)\%$ of the Departments consumption, where $0 < X < 100$. As an example, in 5.3 Department 1.1 and 1.2 are fixed split delegates of Business Unit 2 and as a result their consumption is increased also by the consumption of Business Unit 2. After delegation, Business Unit 2 has its consumption cleared and will not be calculated towards the total Energy Billing.

Due to the possibility of having multiple players of departments, in order to correctly calculate the final bill, the billing process starts from the inner-most tier of departments and iteratively moves outwards until the top-tier departments are reached.
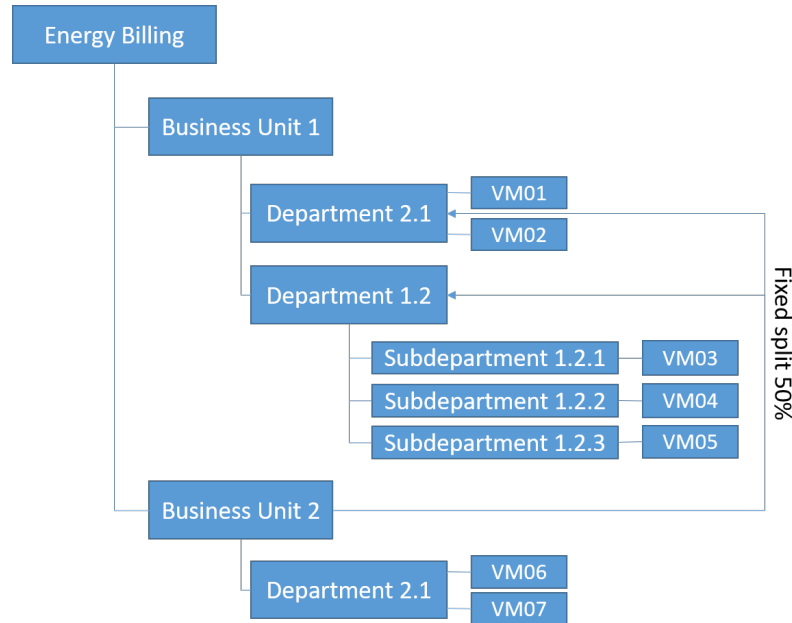
Figure 5.3: Relationship between Departments and Delegates.

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. We follow the design principle that the data stored on the blockchain is the minimal amount of data that needs to be trustlessly verified. Any piece of data that can be derived from the already on-chain data is generated and verified off-chain. In case data does not need to be accessed by another contract but must be transmitted to an end-user, the `Event` functionality of Solidity is used (explained in **??**).

5.4.1 MeterDB

MeterDB utilizes 3.3.2.3 to keep track of the energy consumption values and logged timestamps of a smart meter or a virtual meter. We assume that a meter can be any IoT device that has its own Ethereum account[2]. The contract is designed to map each Meter's address to its details (`id`, `reading`, `timestamp`). As a UX upgrade, a Meter's details can also be fetched by its id, if it is known. The contract is split in two files, `MeterDB.sol` which contains the business logic and `Meter.sol` which contains the logic from 3.3.2.3 that encodes and decodes meter data in a `bytes32` variable. It is designed intentionally to be very light for the purpose of storing and retrieving the latest readings of a meter. For each meter, its latest reading is pulled from the monitoring server and if a new consumption value has arrived it replaces the previous reading along with its timestamp by calling `ping`. Table 5.1 illustrates the design of the storage in `Meter.sol`.

---

[2]A private-public keypair needs to be generated for every meter
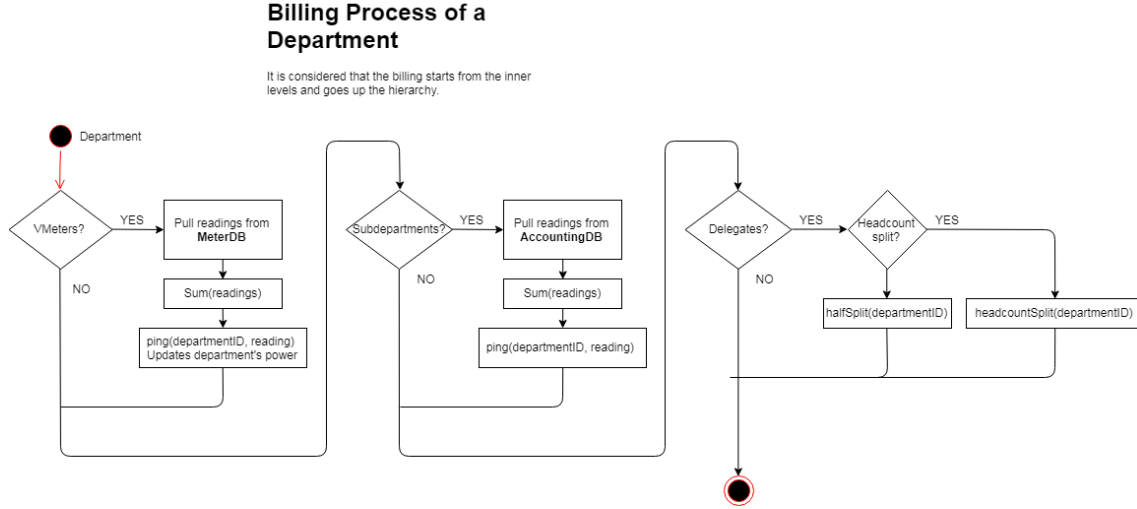
Figure 5.4: Consumption calculation for a Department

Table 5.1: Required variables and size to describe a Meter.

Name	Type	Comment
meterID	bytes8	IDs can be encoded in 8 bytes, can be optimized but hurts UX
creationTime	uint32	Meter supports timestamps up to $2^{**}32 = 02/07/2106$ @ 6:28am (UTC)
power	uint32	Power readings can reach up to $2^{**}32$, which is enough to store 1TWh readings

5.4.2 AccountingDB

Since part of the accounting logic is grouping *Virtual Meters* in *Departments* and storing the Department readings in the smart contact, there is code reuse from `MeterDB.sol`. In addition, we need to keep track of the headcount for each Department, along with its subdepartments, virtual meters, and delegates for further accounting logic. We cannot use Method 3 for encoding the accounting logic because it does not support arrays. As a result, we define two mappings for each Department, one for keeping track of the power consumption of a department and one for maintaining a list of the accounting logic associated with that department. Due to the forwarding of a department consumption to another, we keep track of the power that is generated by the virtual meters, the delegated power that is received from another department, and the cleared power which is the amount of power that has been cleared from that department, either through delegation or from accounting.

The main addition is the forwarding logic. The forwarding can either be a *Fixed Split* or a *Headcount Split*. A fixed split can only be applied to a department with 2 delegates. The first delegate receives X and the second (100-X). In headcount split, for each of the sender's delegates its headcount is received and the receiver's delegated power is increased by the amount of power, proportional to the headcount. At the end, the sender's power is cleared and any further pings to that department do not consider the amount that has already been cleared.

Table 5.2: Required variables and size to describe a Meter.

Name	Type	Comment
active	bool	Department activated?
headcount	uint7	Headcount percentage, between 0 and 100
power	uint32	Power readings can reach up to $2^{**}32$, which is enough to store 1TWh readings
delegatedPower	uint32	Power readings can reach up to $2^{**}32$, which is enough to store 1TWh readings
clearedPower	uint32	Power readings can reach up to $2^{**}32$, which is enough to store 1TWh readings

5.4.3 Access Control

As discussed in Chapter 4 enforcing proper access control on critical functions of a smart contract is very important. It is common to find Access Control Lists (ACL) in enterprise environments which allow only selected participants to have to resources of various levels. This functionality does not exist by default in smart contracts. We proceed to describe the functionality of each of the mentioned roles for our smart contracts:

1. Admin: Has full access to all contract functions

2. Meter Manager: Can enable or disable a Meter at `MeterDB`.

3. Meter: Can call the `ping` function of `MeterDB`

4. Accountant: Can call the billing functions of `AccountingDB`

INITIALLY ALL ACCESS CONTROL WAS DONE INSIDE EACH CONTRACT, HOWEVER IT MADE IT BIGGER TO DEPLOY BLABLA. very light to minimize deployment costs and the costs of calling each function. A third party is able to index all the currently activated meters by monitoring the events that enabled some address to call `ping` on the ACL.

For a meter to be able to ping its reading it first needs to be activated by an authorized party, hereafter referred to as the *Meter Operator*. The *Meter Operator* is able to deactivate the meter, deleting its stored values from the contract.

5.4.4 Contract Registry

As a User Experience upgrade, a `Registry` contract was implemented. The contract links contract addresses to names through a `bytes32` to `address` mapping. The address of a contract can be changed by the registry's operator, which allows for upgradeability for contracts that refer to other contracts. As an example, a contract that calls another contract needs to know its address. Instead of modifying the destination's address in the sending contract, we go through the registry as a proxy.

ADD SHAPE. 1 - 2 - 3.

This can also be used for the client script implementations (??) in order to reduce the number of arguments needed. Instead, the registry's address is passed to each script and they resolve each contract's name to address at runtime.

This pattern can be extended to implement further upgradeable logic, however is considered out of scope.

5.5 Client Side

We utilize Python to interact with the smart contracts and push data to them. We implement multiple command line interfaces which can be used for abstracting the complexity to an operator. The choices for each client script design are further described in Appendix ??

As mentioned in Section 5.3.1, there is no direct access to each meter's reading. As a result, all readings are exposed through an authenticated REST API. We implement a module which gets instantiated with the Meter's Id as a parameter,

and is able to interact with the monitoring server API. The primary function used is `get_single_reading` which fetches 1 reading from the monitoring server, by default the most recent one. In the future, the goal is to have meters which are able to send arbitrary messages over TCP/IP, which would enable this step to be skipped.

We define a base class called `Contract` that provides functionality for interacting with a smart contract. It allows signing offline signing of transactions and broadcasting of raw transactions which is a big security upgrade as the Ethereum node we are submitting to does not need to hold our private keys. It receives a compiled contract ABI and address to interact with the contract along with a private key for signing the transactions. The exposed function `sign_and_send` is used to specify which function of the contract to call, its arguments and submit it for execution to an Ethereum node.

Since these are all low level functionalities, we proceed to describe the higher level scripts which were created to provide the needed functionality of a client. All modules utilize the `Contract`, and Meter clients also use the mentioned Monitoring Server script to pull data from the REST API.

5.5.1 Meter Operator

Wrapper around `Contract` Approves and disapproves meters.

5.5.2 Single and Multiple Meters

Uses monitoring + contract

5.5.3 Single and Multiple Virtual Meters

Uses contract and pulls / pushes. Explain AST parsing

5.5.4 Accountant

Uses 2 contracts, both meterdb and accounting, to do stuff

5.5.5 Administrator

USes the registry/ACL script to manage permissions for who can call what based on the previous things

6 Results

Summarize results from 5, explain how 3 improved scalability and how 4 improved security of the SC's.

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] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] "Colored coins," 2013.

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

[4] N. Szabo, "Smart contracts: Building blocks for digital markets," 1995.

[5] A. Dika, "Ethereum smart contracts: Security vulnerabilities and security tools," 2017.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[7] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," 2004.

[8] "Block hashing algorithm." `https://en.bitcoin.it/wiki/Block_hashing_algorithm`.

[9] Ethereum, "Ethash." `https://github.com/ethereum/wiki/wiki/Ethash`.

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

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[12] "Monero." `https://getmonero.org/`.

[13] "Zcash." `https://z.cash/`.

[14] "Pivx." `https://pivx.org/`.

[15] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," *CoRR*, vol. abs/1107.4524, 2011.

[16] S. Goldfeder, H. A. Kalodner, D. Reisman, and A. Narayanan, "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies," *CoRR*, vol. abs/1708.04748, 2017.

[17] "Eth price stats and information,"

[18] "Opentimestamps: Scalable, trust-minimized, distributed timestamping with bitcoin." `https://petertodd.org/2016/opentimestamps-announcement`.

[19] V. Buterin, "On public and private blockchains." `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains`, 2015.

[20] J. Morgan, "A permissioned implementation of ethereum supporting data privacy." `https://www.jpmorgan.com/country/DE/en/Quorum`.

[21] "Hyperledger." `https://www.hyperledger.org/`.

[22] "R3." `https://www.r3.com/`.

[23] T. T., "Ethereum virtual machine illustrated." `http://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`.

[24] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*, pp. 139–147, Springer, 1992.

[25] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," 12 2013.

[26] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. 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, (New York, NY, USA), pp. 3–16, ACM, 2016.

[27] P. Kasireddy, "How does ethereum work, anyway?," 2017.

[28] E. Hilbom and T. Tobias, "Applications of smart-contracts and smart-property utilizing blockchains," 2016.

[29] "Gas costs from yellow paper – eip-150 revision (1e18248 - 2017-04-12)." `https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mOO9GtSKEKrAsfO7Frgx18pNU/edit#gid=0`, 2017.

[30] "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`.

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

[32] "Irreversible transactions." `https://en.bitcoin.it/wiki/Irreversible_Transactions`.

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

[34] "Bamboo: a morphing smart contract language." `https://github.com/pirapira/bamboo/`.

[35] P. Technologies, "Proof-of-authority chains." `https://wiki.parity.io/Proof-of-Authority-Chains.html`.

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

[37] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," *CoRR*, vol. abs/1703.03994, 2017.

[38] "Raiden network." `https://raiden.network/`.

[39] "Funfair technologies." `https://funfair.io`.

[40] "Raiden network." `https://counterfactual.com/`.

[41] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[42] "Loom network." `https://loomx.io`.

[43] "Cosmos network." `https://cosmos.network`.

[44] V. Buterin, K. Floersch, and D. Robinson, "Plasma cash: Plasma with much less per-user data checking," 2018.

[45] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," 2017.

[46] M. Vukolić, "Rethinking permissioned blockchains," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, (New York, NY, USA), pp. 3–7, ACM, 2017.

[47] Y. Marcus, E. Heilman, and S. 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`.

[48] N. Atzei, M. Bartoletti, and T. 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*, (New York, NY, USA), pp. 164–186, Springer-Verlag New York, Inc., 2017.

[49] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 254–269, ACM, 2016.

[50] ConsenSys, "Mythril." `https://github.com/ConsenSys/mythril`.

[51] "Echidna, ethereum fuzz testing framework." `https://github.com/trailofbits/echidna`.

[52] "Smartcheck." `https://tool.smartdec.net/`.

[53] "Securify." `https://securify.ch/`.

[54] "Zeus: Analyzing safety of smart contracts."

[55] N. Ivica, K. Aashish, S. Ilya, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018.

[56] N. Ivica, K. Aashish, S. Ilya, P. Saxena, and A. Hobor, "Maian." `https://github.com/MAIAN-tool/MAIAN`, 2018.

[57] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[58] A. Ahmad, "Integration of iot devices via a blockchain-based decentralized application," 2017.

[59] M. Thakur, "Authentication, authorization and accounting with ethereum," 2017.

[60] "Grid+." `https://gridplus.io/`.

[61] "Powerledger." `https://powerledger.io/`.

[62] "Brooklyn microgrid,"

[63] E. Mengelkamp, B. Notheisen, C. Beer, D. Dauer, and C. Weinhardt, "A blockchain-based smart grid: towards sustainable local energy markets," *Computer Science - R&D*, vol. 33, no. 1-2, pp. 207–214, 2018.

[64] K. Samani, "Models for scaling trustless computation." `https://multicoin.capital/2018/02/23/models-scaling-trustless-computation/`.

[65] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer, "Decentralization in bitcoin and ethereum networks," *CoRR*, vol. abs/1801.03998, 2018.

[66] V. Buterin and G. Virgil, "Casper the friendly finality gadget," 2017.

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

[68] "Sharding faq." `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`.

[69] F. T. Lorenz Breidenbach, Phil Daian, "Tokenize gas on ethereum with gastoken." `https://gastoken.io`, 2018.

[70] "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/`.

[71] J. Izquierdo, "Library driven development in solidity." `https://blog.aragon.one/library-driven-development-in-solidity-2bebcaf88736`, 2017.

[72] C. Santana-Wees, "Virtualstruct.sol." `https://github.com/figs999/Ethereum/blob/master/VirtualStruct.sol`.

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

[74] H. Jameson, "Faq: Upcoming ethereum hard fork." `https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/`, 2016.

[75] A. Beregszaszi, "Hardfork meta: Spurious dragon." `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-607.md`.

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

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

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

[79] "Ethereum classic." `https://ethereumclassic.github.io/`.

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

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

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

[83] "Standardized ethereum recovery proposals." url-https://github.com/ethereum/EIPs/pull/867.

[84] "Hard fork completed,"

[85] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *CoRR*, vol. abs/1710.06372, 2017.

[86] "New opcode: Staticcall." `https://github.com/ethereum/EIPs/pull/214`, 2017.

[87] "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/.

[88] "Ethernaut." `https://ethernaut.zeppelin.solutions/`.

[89] "Hack this contract." `http://hackthiscontract.io/`.

[90] "Capture the ether." `https://capturetheether.com/`.

[91] E. Comission, "2030 energy strategy." `https://ec.europa.eu/energy/en/topics/energy-strategy-and-energy-union/2030-energy-strategy`.

[92] EY, "Blockchain in power and utilities: real or hype?," 2017.

Appendices

A   Transactions and Blocks

Querying a node for transactions:

```
 1 > web3.eth.getTransaction("0
     x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
     ")
 2 { blockHash: '0
     x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
     ',
 3   blockNumber: 5284738,
 4   from: '0xbc7f7a5eb2039a3330665aa22b5d79485896bd11',
 5   gas: 250000,
 6   gasPrice: BigNumber { s: 1, e: 10, c: [
 7       80000000000
 8     ]
 9   },
10   hash: '0
     x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
     ',
11   input: '0
     x0a19b14a000000000000000000000000000000000000000000000000000000000000
     ',
12   nonce: 9,
13   to: '0x8d12a197cb00d4747a1fe03395095ce2a5cc6819',
14   transactionIndex: 0,
15   value: BigNumber { s: 1, e: 0, c: [
16       0
17     ]
18   },
19   v: '0x1c',
20   r: '0
     xe7df9fee346ea3575dd541e120df7b154d46bba2bf859352b29a7f2c6dd5fab3
     ',
21   s: '0
     x68d266b140ac468ee328592ca6b4a461edace859d6837358bc3f8eeeb24ade63
     '
22 }
```

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

Querying a node for block contents:

```
 1  > web3.eth.getBlock(5284738)
 2  { difficulty: BigNumber { s: 1, e: 15, c: [
 3        32,
 4        85319757566868
 5      ]
 6    },
 7    extraData: '0x7869786978697869',
 8    gasLimit: 7995219,
 9    gasUsed: 1547361,
10    hash: '0
         x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
         ',
11    logsBloom: '0
         x00020800000204000200040000000000000100400000000000080002000000008400080040022
         ',
12    miner: '0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb',
13    mixHash: '0
         x29b6efa55ad0298b0c90f21e9e23d572977ffb3c5064a9816a69bb2bf2a9effd
         ',
14    nonce: '0xabed128000fed25e',
15    number: 5284738,
16    parentHash: '0
         xb7063b9c7b05c95c35a329717e44875829cc740b2e0749e03d54806dcf34b520
         ',
17    receiptsRoot: '0
         xe5e176557b9f40394917191095b706a2a331742f0dc93a10e1d59b5e297ee0b5
         ',
18    sha3Uncles: '0
         x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
         ',
19    size: 7789,
20    stateRoot: '0
         x1c62917ac72a2b76e00053efbb7af0d6949e86cafb3f983812d763715c6c9905
         ',
21    timestamp: 1521484243,
22    totalDifficulty: BigNumber { s: 1, e: 21, c: [
23        31406307,
24        78318927526632
25      ]
26    },
27    transactions: [ '0
         x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
         ',
28        '0
           xbe1c3e767e34d5d668ea50d3400b2e11a663479f931c225eda5e1d314e012589
           ', ...
29    ],
30    transactionsRoot: '0
         xb0a066469d74fe1f450c5fa8a1f59c5b7305feb6336d0d59f347a2b2c7a8c579
         ',
31    uncles: []
32  }
```

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

Ganache UI:
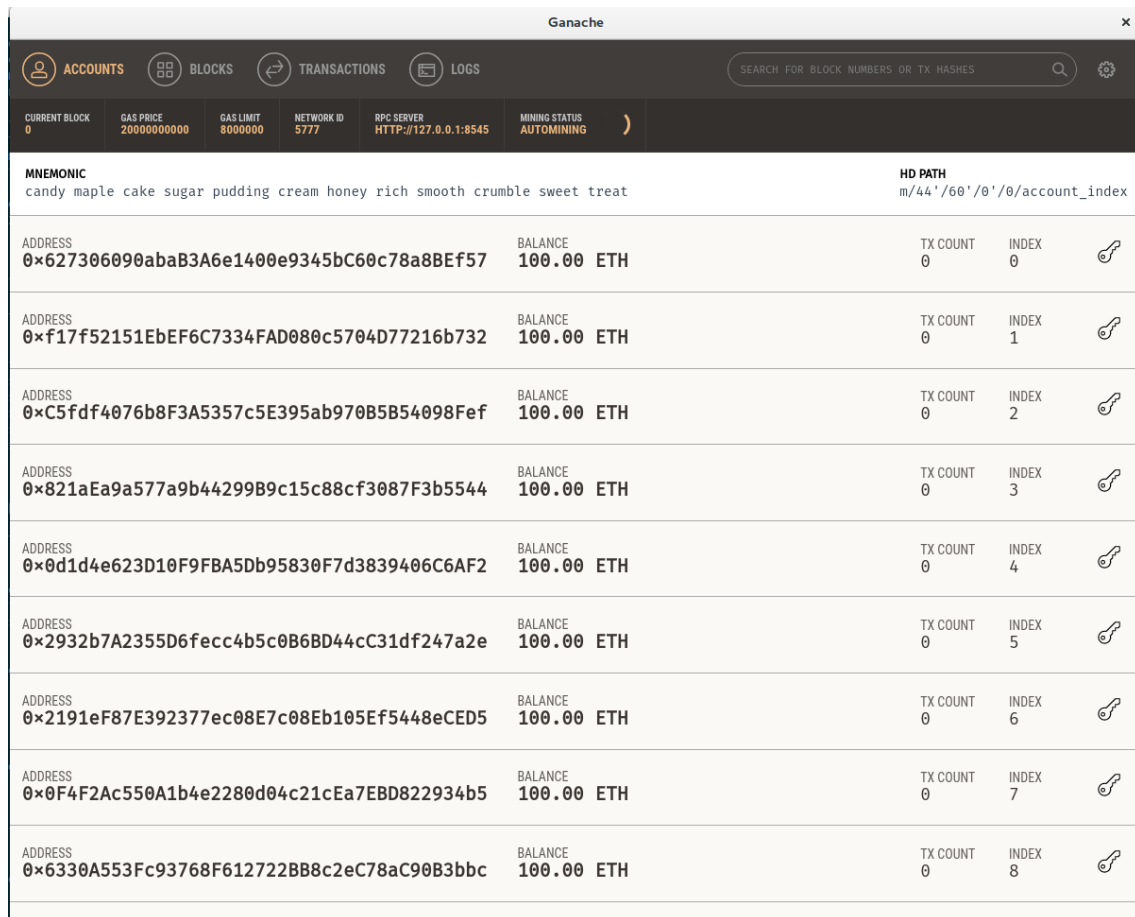
Figure A.3: Ganache testnet User Interface

Web3js example:

```
1 node
2     > Web3 = require('web3');
3 > 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));
4 > web3.eth.blockNumber;
5 5289236
```

Figure A.4: Interacting with a node in Javascript

Web3py example:

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

Figure A.5: Interacting with a node in Python

B   Scalability through Gas Saving masks

```solidity
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 B.1: Example of using the `using X for Y` syntax to enhance operations done on datatypes.

```
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 B.2: Example of using the `using X for Y` syntax to enhance operations done on datatypes.

```
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.3: 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:

```
1  pragma solidity ^0.4.21;
2
3  interface Game {
4      event PlayerRegistered(uint16 playerID, address player);
5
6      function Register() public returns (uint16 playerID);
7      function CreateCharacter(uint256 creationTime, uint256 class,
           uint256 race, uint256 strength, uint256 agility, uint256
           wisdom, uint256 metadata)  external;
8      function GetCharacterStats(uint256 index) external view returns
            (uint16 playerID, uint32 creationTime, uint8 class , uint8
           race, uint16 strength, uint16 agility, uint16 wisdom,
           bytes18 metadata);
9
10 }
```

Figure B.4: Interface for described use-case

Tightly packed code:

```
1    struct Character {
2        uint16 playerID;
3        uint32 creationTime;
4        uint8 class;
5        uint8 race;
6        uint16 strength;
7        uint16 agility;
8        uint16 wisdom;
9        bytes18 metadata;
10   }
```

(a) Character structure definition

```
1    function CreateCharacter(
2        uint256 creationTime,
3        uint256 class,
4        uint256 race,
5        uint256 strength,
6        uint256 agility,
7        uint256 wisdom,
8        uint256 metadata)
9        external
10   {
11       uint16 playerID = player2ID[msg.sender];
12       require(playerID != 0);
13
14       Character memory  c;
15       // Overhead from converting, in order to match interface
16       c.playerID = uint16(playerID);
17       c.creationTime = uint32(creationTime);
18       c.class = uint8(class);
19       c.race = uint8(race);
20       c.strength = uint16(strength);
21       c.agility = uint16(agility);
22       c.wisdom = uint16(wisdom);
23       c.metadata = bytes18(metadata);
24
25       uint CharacterId = Characters.length;
26       emit CharacterCreated(c, CharacterId);
27
28       Characters.push(c);
29   }
```

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

```
 1      function GetCharacterStats (uint256 index)
 2          external view
 3          returns (
 4              uint16 playerID ,
 5              uint32 creationTime ,
 6              uint8 class ,
 7              uint8 race ,
 8              uint16 strength ,
 9              uint16 agility ,
10              uint16 wisdom ,
11              bytes18 metadata
12          )
13      {
14          Character memory  c = Characters [index];
15          return (
16              c.playerID ,
17              c.creationTime ,
18              c.class ,
19              c.race ,
20              c.strength ,
21              c.agility ,
22              c.wisdom ,
23              c.metadata
24          );
25  }
```

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

Figure B.5: Implementation requires a Solidity 'struct' to pack all the variables together. CreateCharacter and GetCharacterStats

Method 2 code:

```solidity
1    function CreateCharacter(
2      uint256 creationTime,
3      uint256 class,
4      uint256 race,
5      uint256 strength,
6      uint256 agility,
7      uint256 wisdom,
8      uint256 metadata)
9      external
10     {
11         uint16 playerID = player2ID[msg.sender];
12         require(playerID != 0);
13
14         uint c = uint256(playerID);
15         c |= creationTime  <<   16;
16         c |= class         <<   48;
17         c |= race          <<   52;
18         c |= strength      <<   56;
19         c |= agility       <<   72;
20         c |= wisdom        <<   88;
21         c |= metadata      <<  104;
22
23         uint CharacterId = Characters.length;
24         emit CharacterCreated(c, CharacterId);
25
26         Characters.push(c);
27     }
```

Figure B.6: Create Character by shifting variables

```
1        function GetCharacterStats(uint256 index)
2            external view
3            returns (
4                uint16 playerID,
5                uint32 creationTime,
6                uint8 race,
7                uint8 class,
8                uint16 strength,
9                uint16 agility,
10               uint16 wisdom,
11               bytes18 metadata)
12           {
13           uint c = Characters[index];
14           return (
15               uint16(c),
16               uint32(c >> 16),
17               uint8((c >> 48) & uint256(2**4-1)),
18               uint8((c >> 52) & uint256(2**4-1)),
19               uint16(c >> 56),
20               uint16(c >> 72),
21               uint16(c>> 88),
22               bytes18(c>>104)
23             );
24       }
```

Figure B.7: 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.8: 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);
```

```
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 );
```

(a) Create Character by shifting variables   (b) get character variables

C   Security

C.0.1   Deleting a struct with mapping

```
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