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

Aristotle University of Thessaloniki

Honda R&D Europe

Georgios Konstantopoulos

Contents

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] (version 0.4.21), the Truffle Framework for streamlining the compilation, testing and deployment process (version 4.1.5), 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*.

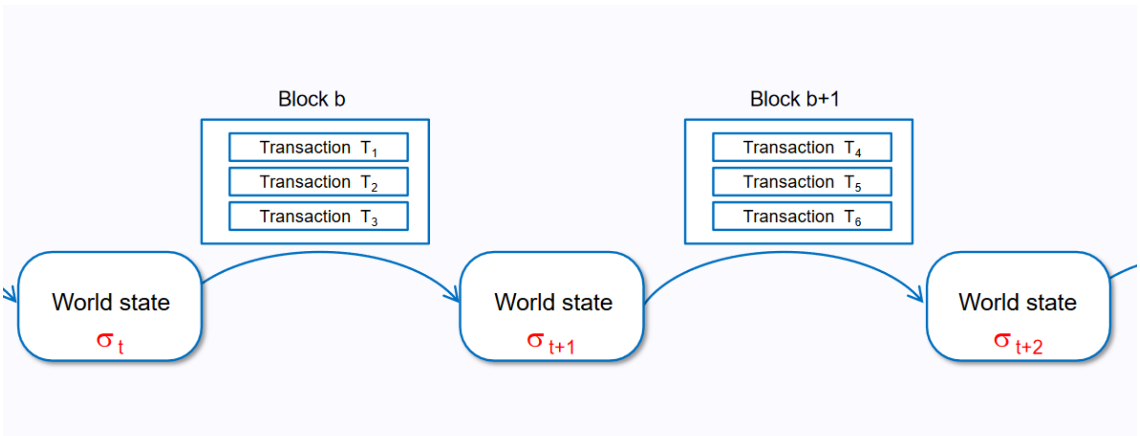


Figure 2.1: Ethereum can be seen as a chain of states [23]

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.

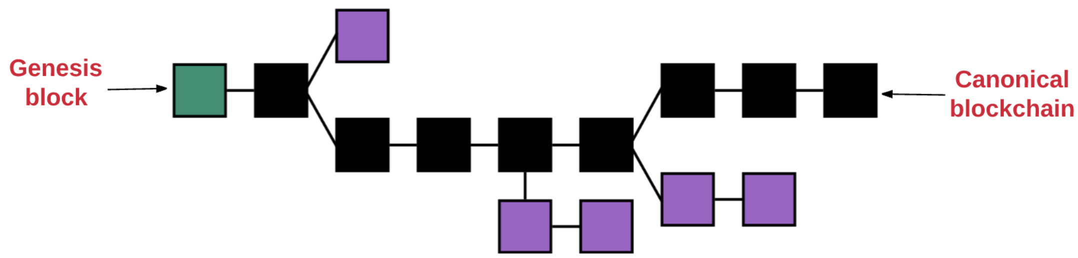


Figure 2.2: The Ethereum protocol chooses the longest chain [27]

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.

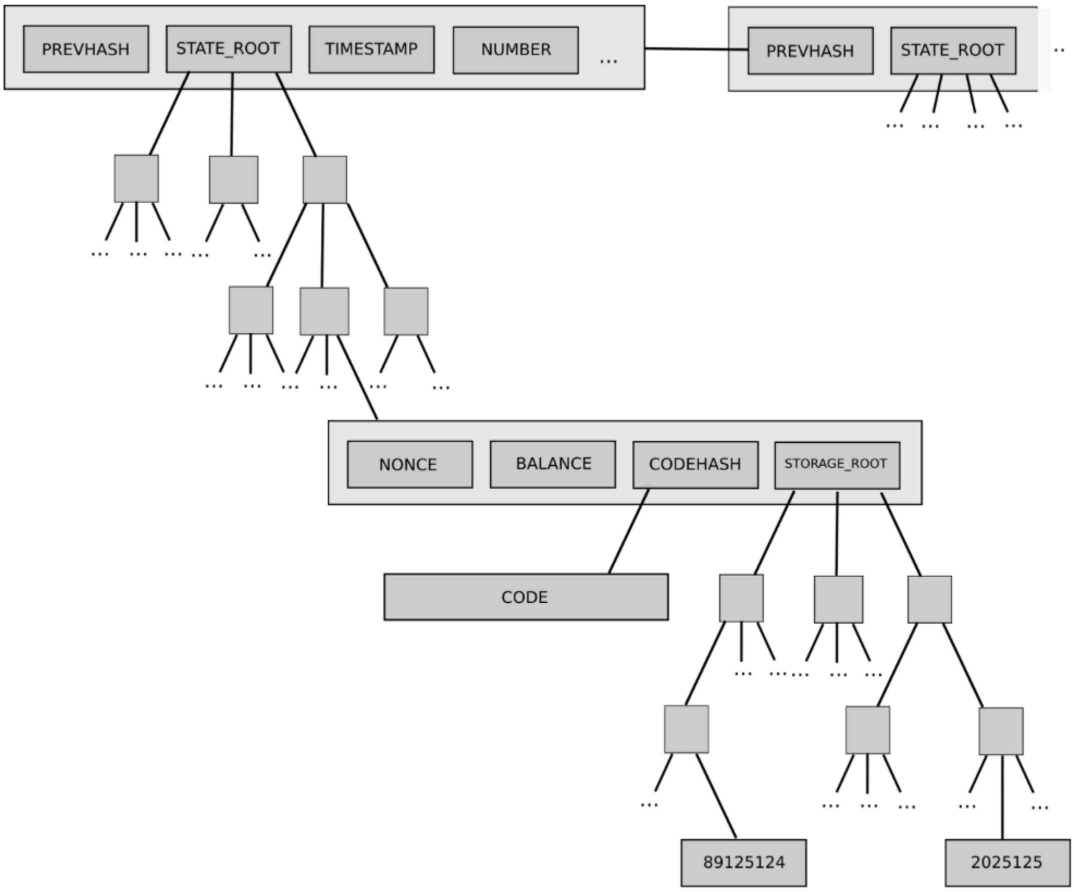


Figure 2.3: The Ethereum world state [3]

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

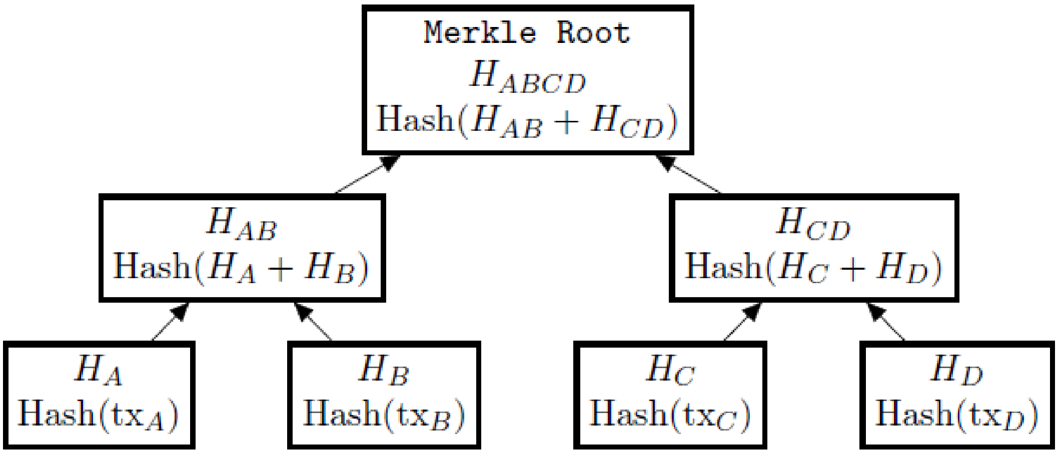
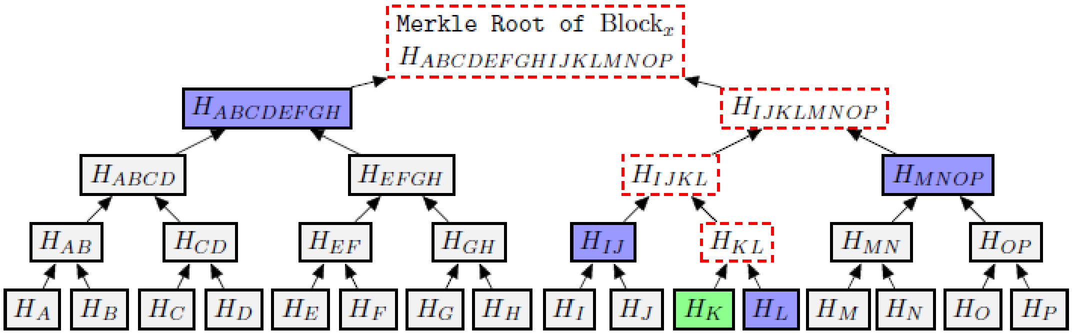


Figure 2.4: Node calculation in a Merkle Tree [28]

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,

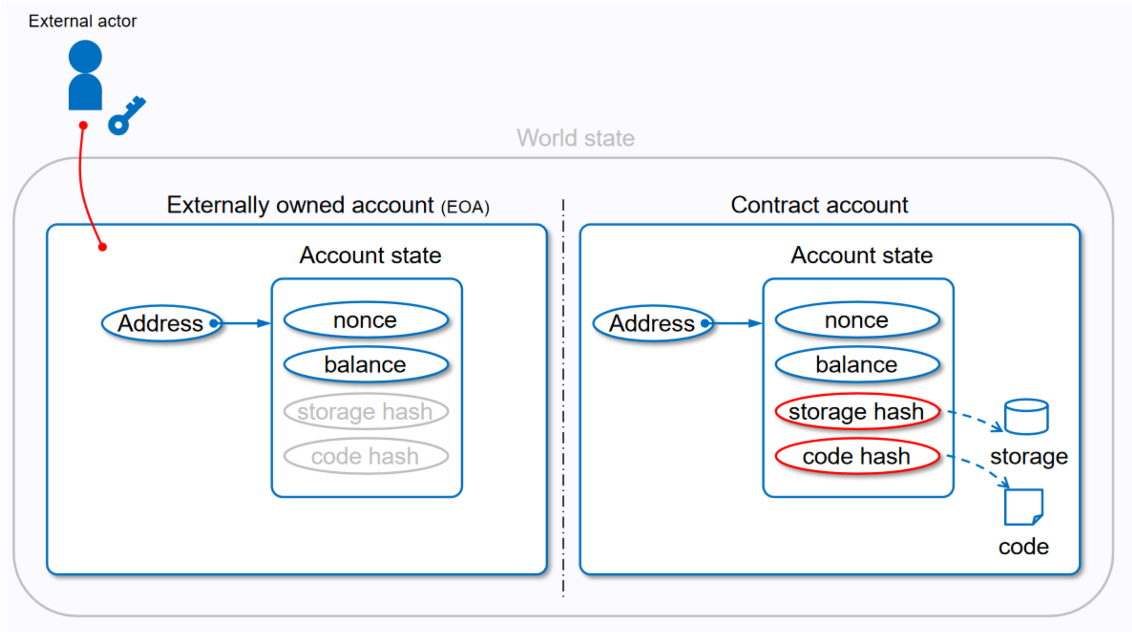


Figure 2.6: Externally Owned Accounts and Contract Accounts in the Ethereum World state [23]

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.

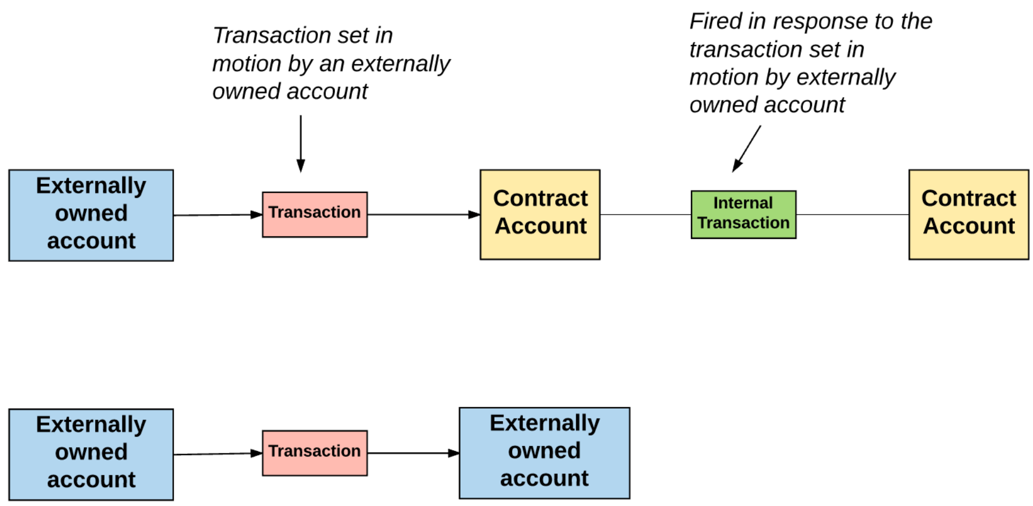


Figure 2.7: Transaction made by an EOA to an EOA or a contract [27]

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:

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

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`

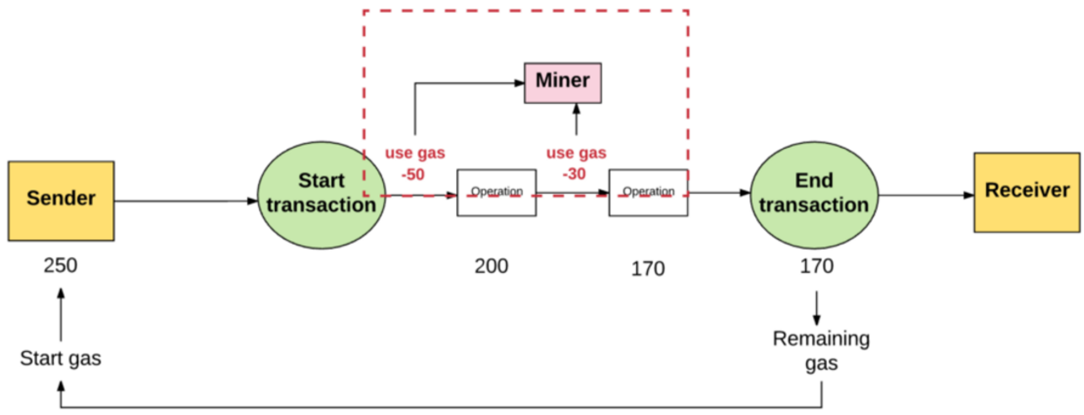


Figure 2.8: Successful transaction [27]

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.

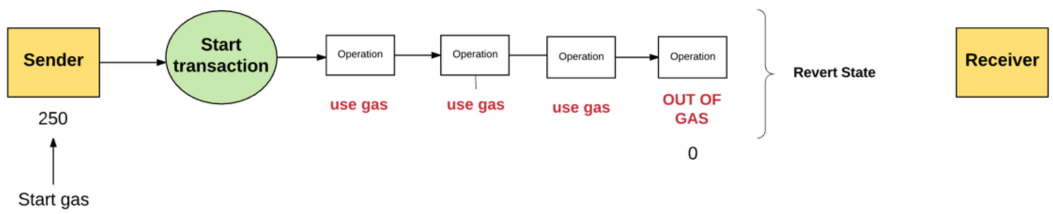


Figure 2.9: Failed transaction that ran out of gas [27]

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 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.

```
pragma solidity ^0.4.16;

contract TestContract {

  string private myString = "foo";
  uint private lastUpdated = now;

  function getString() view external returns (string, uint) {
    return (myString, lastUpdated);
  }

  function setString (string _string) public {
    myString = _string;
    lastUpdated = block.timestamp;
  }
}
```

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].

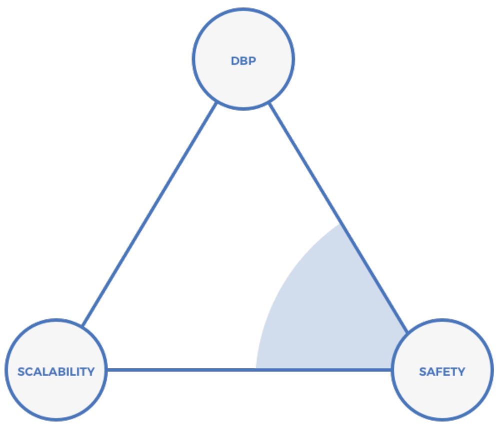


Figure 3.1: The Scalability Triangle [64]

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

As seen in Table 3.1, the most expensive operations involve CREATE[7] and SSTORE[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.

```
Character memory  c;
c.playerID = uint16(playerID);
c.creationTime = uint32(creationTime);
c.class = uint8(class);
c.race = uint8(race);
c.strength = uint16(strength);
c.agility = uint16(agility);
c.wisdom = uint16(wisdom);
c.metadata = bytes18(metadata);
```

(a) Method 1: `CreateCharacter`

```
Character memory  c = Characters[index];
return (
    c.playerID,
    c.creationTime,
    c.class,
    c.race,
    c.strength,
    c.agility,
    c.wisdom,
    c.metadata
);
```

(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 straightforward way. By utilizing Solidity's built-in structures and arrays, we can create an array of `Character` type structures and access their traits by their indexes, as done in B.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.

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

(a) Method 2: `CreateCharacter`

```
return (
    uint16(c),
    uint32(c >> 16),
    uint8((c >> 48) & uint256(2**4-1)),
    uint8((c >> 52) & uint256(2**4-1)),
    uint16(c >> 56),
    uint16(c >> 72),
    uint16(c>> 88),
  bytes18(c>>104)
);
```

(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

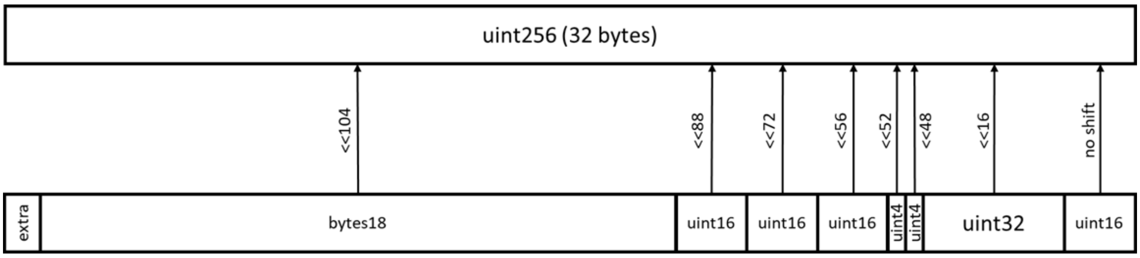


Figure 3.4: Encoding data in a `uint256` variable

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.

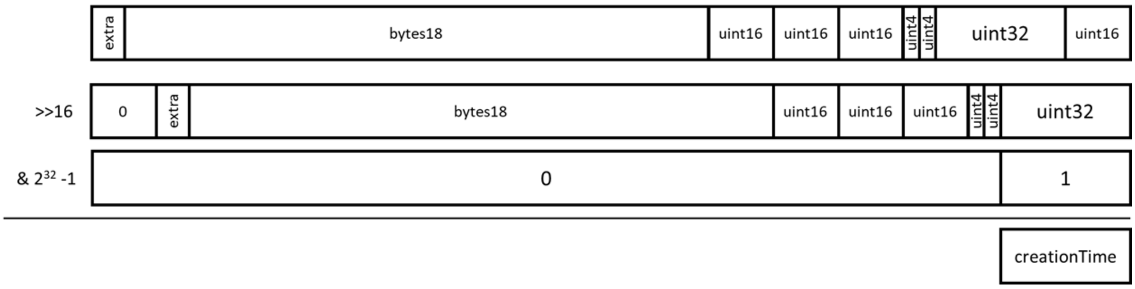


Figure 3.5: Retrieving `creationTime` from the encoded `uint256`

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.

```
bytes32 c = c.SetPlayerID(playerID);
c = c.SetCreationTime(creationTime);
c = c.SetClass(class);
c = c.SetRace(race);
c = c.SetStrength(strength);
c = c.SetAgility(agility);
c = c.SetWisdom(wisdom);
c = c.SetMetadata(metadata);
```

(a) Method 3: `CreateCharacter`

```
bytes32 c = Characters[index];
return (
    c.GetPlayerID(),
    c.GetCreationTime(),
    c.GetClass(),
    c.GetRace(),
    c.GetStrength(),
    c.GetAgility(),
    c.GetWisdom(),
    c.GetMetadata()
);
```

(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].

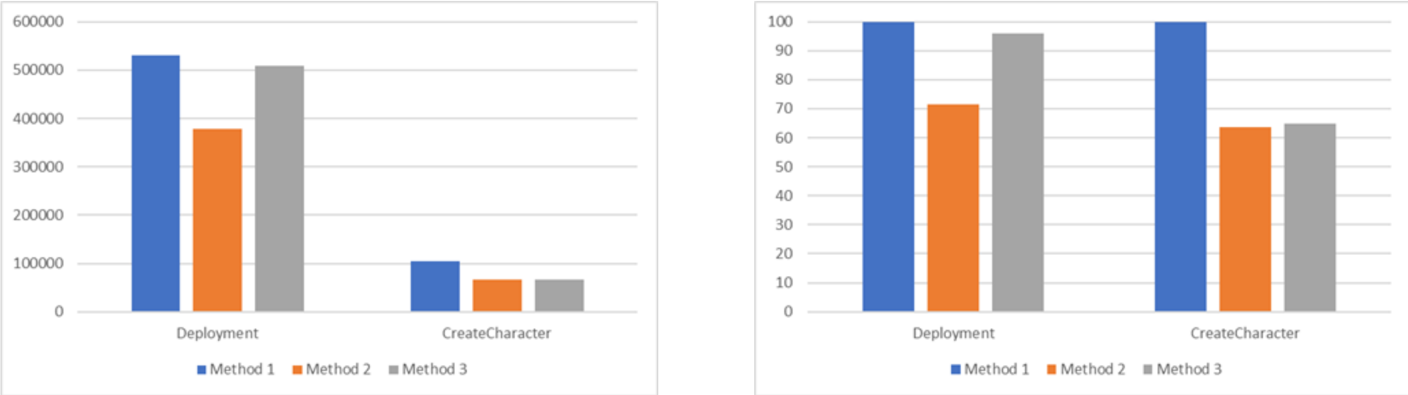


Figure 3.7: Gas cost comparison between the 3 proposed methods.

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 implementation of the protocol, the security of smart contracts and the best practices that need to be applied in order to code secure and robust smart contracts. We contribute to the existing literature by evaluating the auditing tool *Slither* and comparing it to the performance of other automated auditing tools, which can detect 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 large amounts of data, which resulted in clients taking long to synchronize, effectively causing a *Denial of Service* attack to the network [74]. As a response, two hard-forks[1] were proposed. The first one, Tangerine Whistle[75] solved the gas pricing issue and at a later point, Spurious Dragon[75] cleared the world state from the accounts created by the attack.

Eclipse Attacks on Ethereum Eclipse attacks are a type of attack in which an attacker floods a victim node with TCP connections, allowing the attacker to control what blocks are received by the victim. The victim sees a different blockchain than the valid one and this can result in double-spend attacks against vendors who use victim nodes as sources for the state of the blockchain. This is a known and fixed attack on Bitcoin which requires an attacker to control a large number of nodes with unique IP addresses in order to monopolize the outgoing and incoming connections to the victim [76]. On the other hand, Eclipse attacks on Ethereum require only a small number of attacking nodes, because it uses a structured network protocol contrary to Bitcoin's unstructured network protocol which makes random connections [47]. The attack vector was communicated to the client developers and the vulnerabilities

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

were fixed in *geth* v1.8[2]. 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 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 been exploited multiple times the past. We proceed to give a brief explanation of the three biggest hacks in Ethereum smart contracts, the 'TheDAO' and the 'Parity Multisig 1 and 2'[3] which involved two independent incidents with the same smart contract.

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 [77].

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*[4] attack which allowed an attacker to make unlimited withdrawals from the contract [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, which was 12% of Ether in existence at the time [77]. This was not accepted by the whole community, and as a result, users who decided not 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 Section 3.3.2.3, involves using the *proxy libraries* pattern 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 [81].

[2]Geth (go-ethereum) is the most popular implementation of the Ethereum protocol in golang

[3]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 widely used by organizations to decrease the chances of funds being stolen as well as to ensure that funds get spent only when all authorized parties agree. The Parity Multisig refers to a multisig wallet implementation by Parity Technologies, `https://www.parity.io/`

[4]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.

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, however due to it being unprotected it was callable by any user of the wallet[5]. As a result, a malicious user could reinitialize any multisig with their address as the contract owner and drain the multisig funds.

This was observed by a group of ethical 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 [82]. The primary cause of this hack was due to the irresponsible use of `delegatecall` and the lack of proper access control on the `initWallet` function, along with the lack of external auditing of the wallet contracts before deploying to mainnet.

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, patching the vulnerability which caused the previous hack. The fix in `initWallet` involved adding a `only_uninitialized` modifier which would only allow modification of the linked multisignature wallet owners during initialization[6]. However, the `initWallet` function was never called on the library contract itself, leaving the library uninitialized. As a result, any user could call the `initWallet` function and set themselves as the owner of the library contract. This alone would not have been dangerous, had there not been a `kill` function in the smart contract, which when called deletes the contract's bytecode, and effectively renders it useless.

The attacker first 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 belonging to each project [83].

A number of proposals were made in order to recover the locked funds [84, 85]. All of these would require an 'irregular state change' similar to what happened with the DAO and have caused controversy among the ethereum community, before being eventually dismissed.

4.2 Evaluating Smart Contract Security

Due to the high financial amounts often involved with smart contracts, there need to be guarantees that smart contracts are sufficiently secured as well as execute code as intended. Security audits from internal and external parties to the development team are considered a necessary step before deployment to production. It is also common practice for projects to 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 as well as taxonomies aiming to organize past smart contract vulnerabilities are being

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

[6] `https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`

released over time, but due to how often new vulnerabilities surface, they are insufficient to create a whole picture of the smart contract security ecosystem [?, 48, 5].

In order to improve the security of deployed smart contracts, developers become accustomed to using automated auditing tools as well as use the latest versions of basic tools like the Solidity Compiler. As an example, none of the tools mentioned in [5] were able to detect the 'Uninitialized Storage Pointer' vulnerability[7], 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 the literature, we came in contact with TrailOfBits, a security auditing firm, and used their suite of tools to extend the already built taxonomies [86]. We utilized the tool Slither to audit smart contracts. We process all the smart contracts with the latest version of the Solidity compiler, in order to get the most updated list of warnings and errors.

Slither is a static analyzer and operates on the source code of a smart contract. Its modules (called 'detectors') are able to find certain coding patterns which can be considered harmful when present. This includes detecting popular past contract vulnerabilities such as reentrancy or the 'Parity bugs'. It should be noted however that it is not able to perform symbolic execution and go through all the possible states of a smart contract like Oyente or Mythril [49, 50]. We describe the currently supported detectors by Slither:

Constant/View functions that write to state: Detects `view` functions that modify a smart contract's state. A `view` function uses the `STATICCALL` opcode and as a result is unable to modify the state of a smart contract[87]. This is not enforced by Solidity.

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 additional 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 a contract, leading to theft 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[8] issues. 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[9]. This has not been exploited in the wild, however it can be critical in the case of a banking DApp that keeps track of its users' balances . A full Proof of Concept is given in Appendix C.0.1

[7] `https://github.com/ethereum/solidity/issues/2628`
[8] `http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/`
[9] This is warned for in `http://solidity.readthedocs.io/en/v0.4.21/types.html`

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 state variables of the parent can only be modified by the functions of the parent (the same applies for the child). This has been exploited in smart contract honeypots and is currently undetectable by other automated security tools.

Unprotected Function Detection: If the visibility modifier of a function is 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 can be used to ensure the implementation of an interface matches its 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[10], using 'emit' for events is going to be mandatory from Solidity 0.5.0 and onwards.

Slither can be used both for finding known vulnerabilities, but also to avoid common coding 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[11]. `block.blockhash` has been misused in smart contracts and has resulted in theft of funds [88]. 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, with their true goal being to steal the funds of potential hackers. Hackers who attempt to exploit them need to first deposit some amount of ether 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

[10] `https://github.com/ethereum/solidity/issues/2877`

[11] `http://solidity.readthedocs.io/en/v0.4.21/security-considerations.html`

stolen amounts. We proceed to describe the functionality of a honeypot which takes advantage of 'Variable Shadowing' in Solidity.

The contract in Figure 4.1 was deployed with an initial amount of 0.1 ether as a balance. At first, it seems that an adversary can send an amount of ether to the smart contract, and due to the fallback function[12], if the amount is larger than the value of `jackpot` the `owner` variable will be changed to the attacker's address. After that, the adversary should be able to call `takeAll` and withdraw the funds from the contract.

```
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.

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[13] which looks for the value of `owner`

[12] A fallback function is a unnamed function which gets called when a call is made to a smart contract that does not match any of its function signatures, or when ether is send to the contract's address.

[13] A modifier is a special function which gets executed before or after the function it *modifies*.

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

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.

```
[~/Diploma/security/tools/slither, master]: ./slither.py ../../honeypots/known-honeypots/shadowing-state-variable/KingOfTheHill.sol --disable-solc-warnings
INFO:Slither:Missing constructor in ../../honeypots/known-honeypots/shadowing-state-variable/KingOfTheHill.sol, Contract: KingOfTheHill (owner)
INFO:Slither:Shadowing in ../../honeypots/known-honeypots/shadowing-state-variable/KingOfTheHill.sol, Contract: KingOfTheHill, Contract/Vars {u'Owned': set([u'owner'])}
```

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, like OpenZeppelin [89], ensures that these parts of the code are going to be secured.

Developers must be familiar with the security best practices[14], 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 [90, 91, 92]. 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 [93].

They are primarily used to enforce access control in functions.

[14] `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 the usage of distributed energy resources (DERs) [94]. With the integration of DERs, consumers are gradually transforming to *prosumers* who store their energy surplus or sell it to their peers, effectively decentralizing 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 [95]:

1. **Transaction complexity:** The complexity of energy transactions is correlated to the size of the network participants

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

3. **Empowered prosumer:** Monitoring tools and infrastructure does not allow prosumers to manage their energy production and consumption sufficiently

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 logistics:** 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:** Micropayments can occur when there are short term coincident of wants, such as in the case of charging a car in a traffic light stop from the energy surplus of another car. The supplier is expected to receive a micropayment for the energy they supplied. Using a blockchain for the payment is more efficient than a centralized payment provider due to speed of confirmation and low transaction costs

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.

5.3.1 Metering of Energy

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

A *Virtual Meter* is a group of *Smart Meters*. The purpose of a Virtual Meter (VM) is to track the consumption of multiple Smart Meters across various rooms

[1]Hereafter, refers to a kWh,timestamp tuple which gets stored in the blockchain.

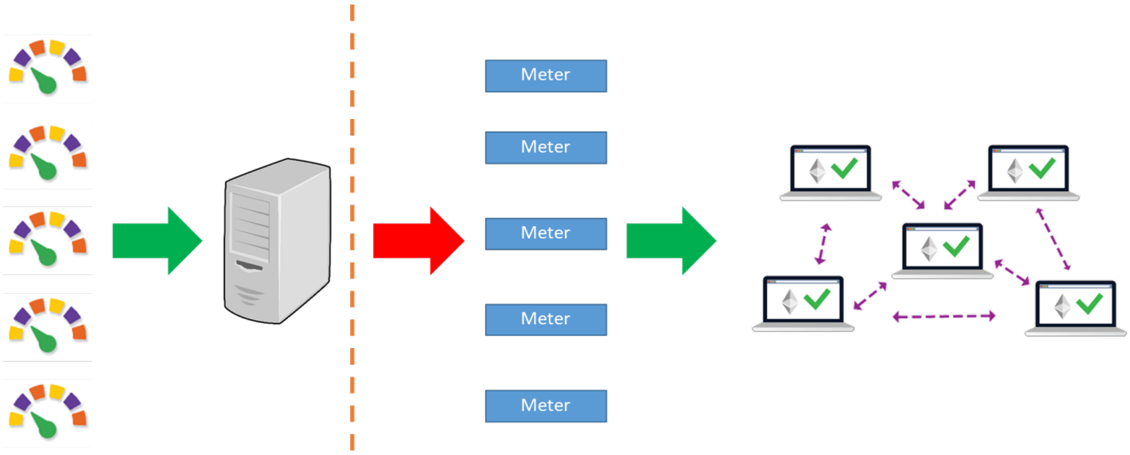


Figure 5.1: Meter Data gets pushed to the blockchain after getting pulled from the monitoring server

of a company building. The consumption of a VM is a function of the power consumptions of its associated smart meters.

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

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

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, as shown in Figure 5.2.

5.3.2 Accounting Logic

The accounting business logic of the company breaks down in a hierarchy which is described by *Business Units* (BUs) and *Departments*. BUs are used for external accounting, while Departments are used for internal accounting. In implementation terms, a Business Unit is equivalent to a Department.

A *Department* is composed of *Virtual Meters*, other Departments of a lower tier (hereafter called *Subdepartments*) and *Delegates*. A Department may forward its

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

power consumption to its Delegates as part of the internal accounting process. The total consumption of a department is the sum of energy consumed by its Virtual Meters, Subdepartments and Delegates. After a department delegates its debt, its consumption is cleared.

There are two ways for a department to forward its consumption to its delegates:

1. **Headcount Split:** A department can perform a headcount split of its consumption based on the percentage of staff that works at each of its *Delegate* department.

2. **Fixed Split:** A department with exactly two *Delegates* can perform a fixed split of its consumption, where thefirst delegate receives $X\%$ of the department's consumption and the second delegate receives $(100 - X)\%$.

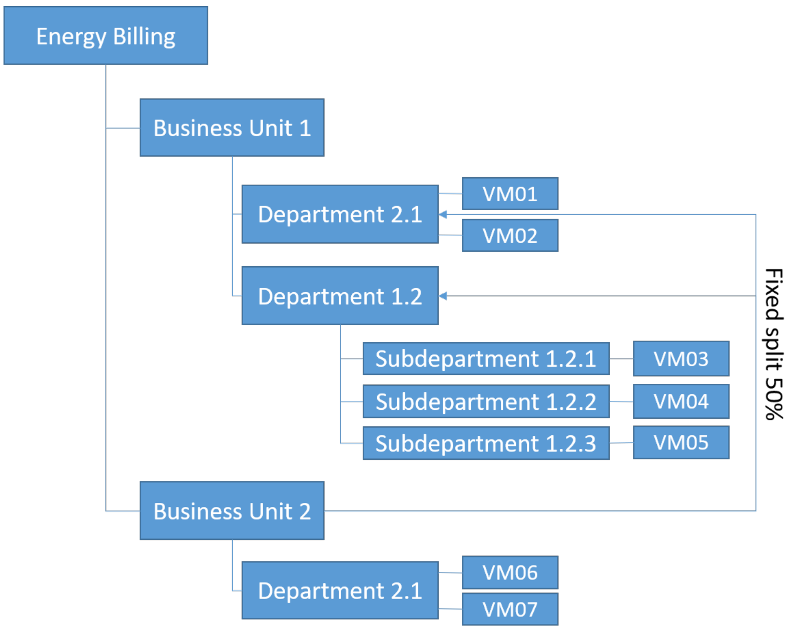


Figure 5.3: Relationship between Departments and Delegates.

As an example, in 5.3, BU2 performs a fixed split and forwards its consumption to its 2 delegates, Department 1.1 and Department 1.2. After delegation, the consumption of Business Unit 2 is cleared and is not calculated towards the total Energy Billing.

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, a Solidity `Event` is emitted (explained in D.1). It is considered that selected functions of each smart contract must be only used by selected operators, and thus they must be protected by an access control mechanism.

5.4.1 MeterDB

MeterDB utilizes Method 3 (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`, `lastPower`, `lastTimestamp`, `power`, `timestamp`).

The contract is split in two files, `MeterDB.sol` which contains the business logic, and `Meter.sol` which contains the gas optimization logic that encodes and decodes meter data in a `bytes32` variable. In order for a meter to be able to call the `ping` function it first needs to be activated, by registering its id and address in the smart contract. Deactivating a decommissioned meter results in its data being deleted from the storage of the smart contract, which refunds gas as explained in Chapter 3. We additionally store a mapping of `id`s to addresses to access a meter in $O(1)$ time instead of iterating over a list of registered meters and checking if the `id` is matched.

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`. A `Pinged` event is emitted after each `ping` function call which enables users to fetch the full history of readings and timestamps for a meter. Table 5.1 illustrates the design of the storage in `Meter.sol`. A timestamp is made to count up to 2^{32} which is big enough to handle time values until the year 2106. Power readings can be up to 2^{48} which allows meter values up to multiple terawatthours, with milliwatthour granularity.

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

Name	Type	Comment
meterID	bytes8	A Meter's ID can be stored in 8 bytes
lastPower	uint48	Previous power reading, up to 2**48
lastTimestamp	uint32	Previous timestamp, up to 2**32
power	uint48	Current power reading, up to 2**48
timestamp	uint32	Current timestamp, up to 2**32

[2]As a prerequisite, a private-public keypair needs to be generated for every meter.

5.4.2 AccountingDB

AccountingDB stores the data required to performing metering and billing for a department. Each department's data is broken down in department data and its accounting data. Storing department data involves using Method 3 (3.3.2.3) to store the consumption of each department, the headcount and activity status of the department and is done in `Department.sol`. For each department, the list of Virtual Meters, Subdepartments and Delegates is stored in a `struct`, because Method 3 cannot be used for array types.

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 an accountant. The consumption forwarding logic is implemented as described in Section 5.3.2.

Table 5.2: Required variables and sizes to describe a Department.

Name	Type	Comment
active	bool	Department active flag
headcount	uint7	Headcount percentage, between 0 and 100
power	uint80	Department power from its virtual meters, up to 2**80
delegatedPower	uint80	Department power that has been delegated to it, up to 2**80
clearedPower	uint80	Department power that has been cleared, up to 2**80

5.4.3 Contract Registry

In order to be able to retrieve the addresses of smart contracts by name, a `Registry` smart contract was implemented. The contract resolves contract names to addresses through a `bytes32` to `address` mapping. The address of a contract can be changed by the registry operator. This lays the foundation of a basic pattern for upgradeable smart contracts via a controlled name registry. Exploring the possible implementations of upgradeable contracts with this method is considered out of scope for this Thesis. Finally, this method can be utilized in the client scripts in Section 5.5 to reduce the number of addresses that need to be supplied to interact with the smart contracts[3].

5.4.4 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 participants to access selected resources based on their clearance levels. This functionality does not exist by default in smart contracts. Access control can be done separately in each contract, by implementing whitelists and checking that a sender of a transaction is whitelisted, however that needs to be done for each contract and does not scale for a rich smart contract ecosystem (further explained in D.2).

We use a pattern for access control that involves calling an external smart contract that stores the permissioning logic. We proceed to describe the access control enforced on each of the identified roles involved in the developed smart contracts:

[3]The registry's address is provided from the command line and then the registry resolves each contract's address by their names which are known beforehand

1. Administrator: Can call all functions in all deployed contracts.
2. Registry Operator: Can call `enable` and `disable` in `Registry`.
3. Meter Operator: Can call `activateMeter` and `deactivateMeter` in `MeterDB`.
4. Accounting Operator: Can call `activateDepartment`, `deactivateDepartment`, `setDelegates`, `setVirtualMeters`, `setSubdepartments`, `setHeadcount` in `AccountingDB`.
5. Accountant: Can call `billPower`, `fixedSplit`, `headcountSplit` and `clearDepartment` in `AccountingDB`.

5.4.5 Deployment Pipeline

The full deployment pipeline involves deploying each contract separately and linking it to the Access Control List as shown in Figure 5.4. At each step, the Administrator grants permissions to call specific functions of the smart contract to each role. Finally, after all contracts have been initialized, their addresses are stored in the Registry for later resolving.

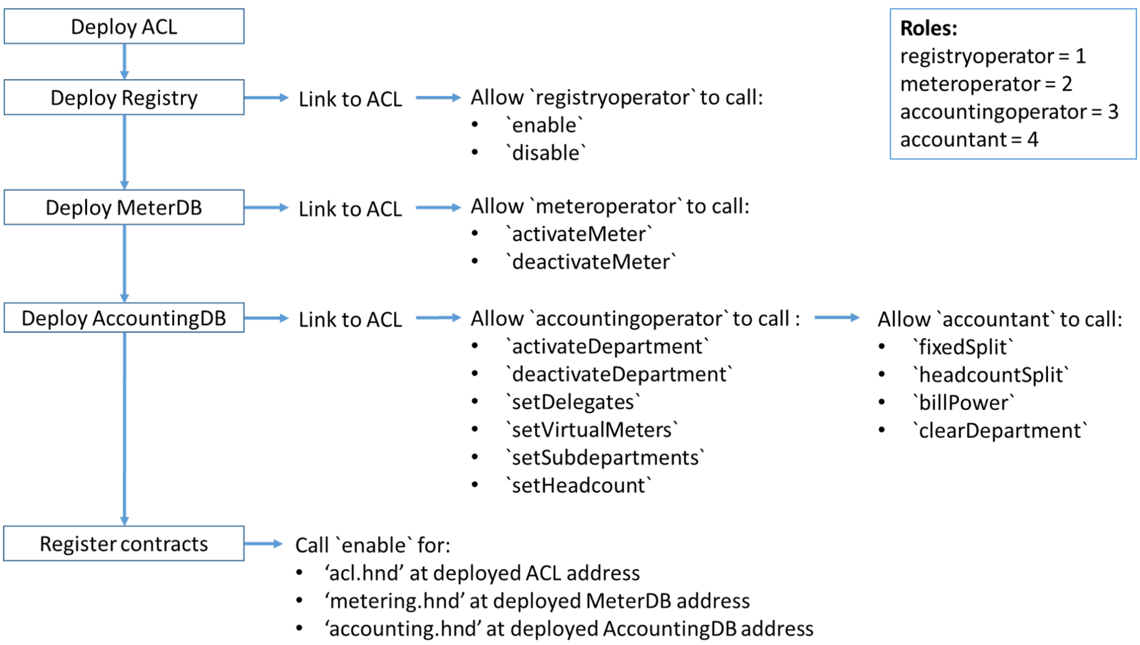


Figure 5.4: The full deployment pipeline enforcing access control

5.5 Client Side

As described in Section 2.3, smart contract functions get executed only in response to transactions made by EOAs. Since the goal is to have an automated process of pulling data from and pushing data to the smart contracts, we implement client scripts which perform that functionality. We use Python and Web3.py to implement utilities and multiple command line interfaces (CLIs) which are used to interact with the smart contracts. A module is also developed for interacting with the

REST API of the monitoring server. A separate keypair is generated for every role mentioned in Section 5.4.4, and each CLI is designed to function only with by using the corresponding role's private key.

5.5.1 Utilities

Web3.py and Contract Wrappers

We created a wrapper around the web3 library in order to provide a generic interface when connecting to a node. That way, a web3 instance can be obtained by calling `getWeb3(endpoint)`, where `endpoint` is the Ethereum node IP and RPC port. Generally, in order to send a transaction, it must first be signed with the sender's private key. When a user hosts their own node, they can store their private key encrypted on the node and unlock the account when they want to send a transaction. In the case where multiple clients need to connect to a third-party node, there is a need to be able to send transactions that are already signed, without having to expose the private keys to the node.

We define a base class called `Contract` that provides functionality for interacting with a smart contract. It allows signing transactions locally and broadcasting them to an Ethereum node. It is able to track the nonce of an account which means that an account can submit multiple transactions without waiting for the previous one to get confirmed. During initialization, it receives the ABI and address of the corresponding 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 and submit it for execution to an Ethereum node. We use the `Contract` class throughout our implementation to create user-friendly interfaces for the functions of each contract while maximizing code reusability and security by signing transactions locally.

Monitoring Server and Rest API

As mentioned in Section 5.3.1, there is no direct access to each meter's reading, which forces us to access all readings through a REST API. We implemented a module which gets instantiated with a Meter 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). Implementing the whole set of functionality provided from the API was considered out of scope becaues it did not provide any additional information for our use-case.

5.5.2 Administrator CLI

The administrator role is used to grant or revoke role permissions to operators. The CLI interacts with the ACL smart contract which has been initialized as described in Section 5.4.4. As input, it expects an action (grant, revoke) and a json configuration file with the address of each account and its corresponding role to be granted or revoked. If no configuration file is supplied, it is expected that a role and address are provided as command line arguments. A user can additionally supply the `fund` parameter to send ether for the gas costs the operators will incur for interacting with their corresponding smart contracts.

5.5.3 Registry Operator CLI

The registry operator's role is used to enable or disable a contract in the `Registry` contract. It can be utilized to register a new contract name and address at a later point in time or to remove an existing contract from the listing. The CLI also provides the functionality to resolve a contract name to its address or a contract address to its name.

5.5.4 Meter Operator CLI

The meter operator role is used to activate or deactivate smart meters in the `MeterDB` contract. A meter can only call the `ping` function only if it is activated by the operator. The CLI expects as input a json configuration file with the address of each meter and its id. If no configuration file is supplied, it is expected that an address and a meter id are provided as command line arguments. If the `deactivate` parameter is given, the CLI tries to deactivate the meters in the list that are already activated. The operator can additionally supply the `fund` parameter to send ether for the gas costs the meters will incur for calling `ping`.

5.5.5 Accounting Operator CLI

The accounting operator role is used to activate or deactivate departments in the `AccountingDB` contract. After activation, the operator sets each department's headcount, virtual meters, subdepartments and delegates according to the business logic supplied in a json configuration file that must be supplied as a command line argument.

5.5.6 Single and Multiple Meters CLI

A `meter` module is implemented which is responsible for connecting to `MeterDB` and the monitoring server API as explained in Section 5.5.1. The meter first needs to be activated by the meter operator as explained in Section 5.5.4. The client is responsible for polling the monitoring server and ping any new power readings to the smart contract. The `single-meter` CLI expects the meter's private key as input, and retrieves the meter id from the smart contract in order to connect to the monitoring server.

Due to having multiple smart meters in our use-case, a `multiple-meters` CLI is also implemented which is responsible for launching multiple `single-meter` processes, each one running independently of each other. The output of each process is saved in a log file.

5.5.7 Single and Multiple Virtual Meters CLI

A `virtualMeter` module is implemented which is responsible for connecting to `MeterDB` and pulling the reading of its connected meters as described in Section 5.3.1. Along with the virtual meter's private key, a function is supplied to the CLI as arguments which specifies how the Virtual Meter consumption is calculated from its associated meters. The virtual meter and the meters included in the formula must be previously activated by the meter operator. Given the formula, each referenced meter's readings are fetched, and the formula is evaluated[4] before pinging the final value to the smart contract.

Similarly to normal smart meters, a `multiple-virtualmeters` CLI is implemented which is responsible for launching multiple `single-virtualmeter` processes, each one running independently of each other. The output of each process is saved in a log file.

5.5.8 Accountant CLI

The accountant module is responsible for pulling all the readings associated with a department, calculating the total amount of energy to be billed to the department as well as performing any forwarding of energy to a department's delegates. The accountant is simultaneously connecting to `MeterDB` and `AccountingDB`. Even though the accountant role is not authorized to access the functions related to pinging or operating in `MeterDB`, accessing the data for each meter can be done without supplying the contract instance with a private key[5]

The departments are processed in order from the inner-most tiers towards the top level departments, as described in Section 5.3.2. The flowchart in Figure 5.5 shows the process that each department goes through in order to correctly calculate its final consumption.

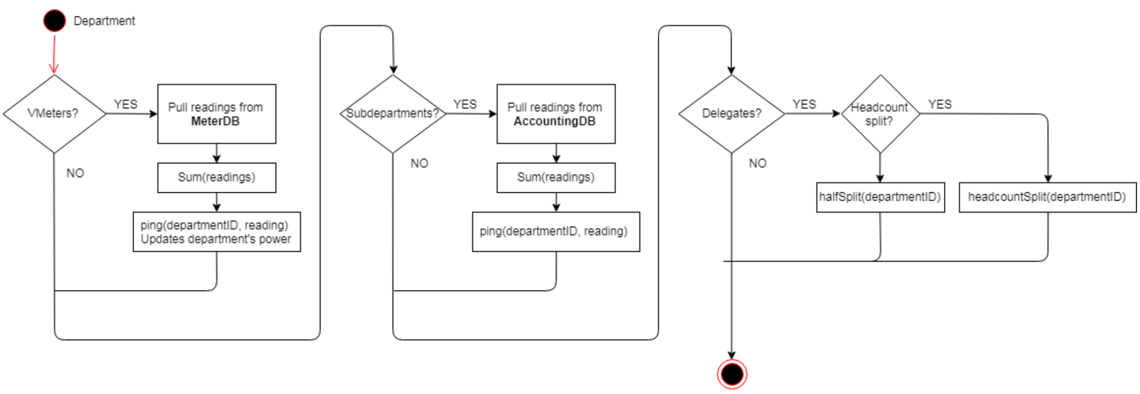


Figure 5.5: Consumption calculation for a Department

5.5.9 Execution Pipeline

Having described how each module of the client side works, in Figure 5.6 we show the order in which the client side programs get executed to provide the desired metering and billing functionality. Firstly, the administrator grants all of the needed privileges to the corresponding operators' addresses. Then, the meter operator activates all meters, and all virtual meters. The accounting operator activates and initializes the accounting settings for each department. Each meter and virtual meter starts pushing their values asynchronously to the `MeterDB` contract, while at the same

[4]Using the module: `https://github.com/Axiacore/py-expression-eval`

[5]Viewing data stored in a smart contract is free as it involves directly querying the node for its storage and does not involve any state mutating transaction. As a result, no private key is needed to access the meter readings.

time the accountant is able to bill each department according to the logic which was previously specified by the accounting operator.

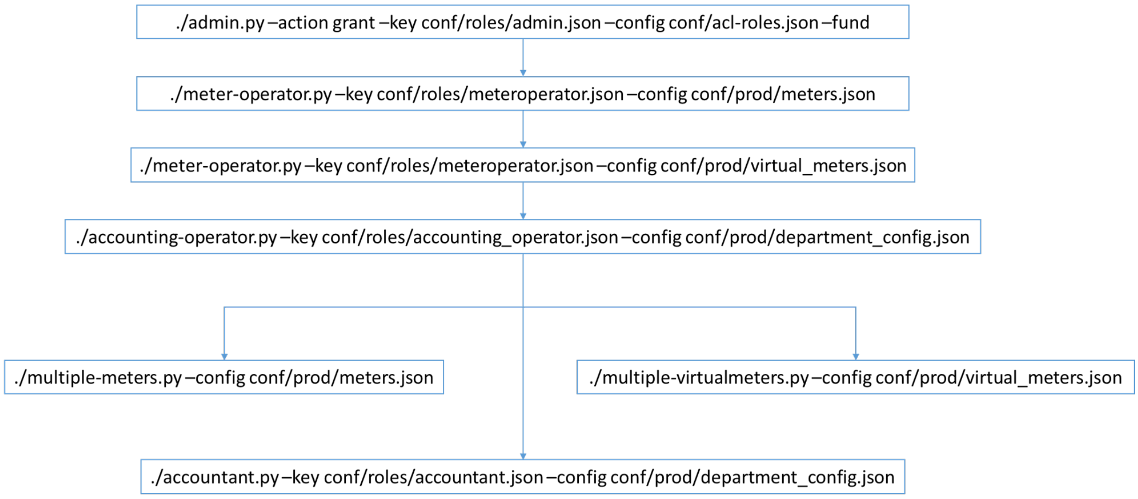


Figure 5.6: The full execution process of metering and billing

6 Evaluation of Implementation and Results

We have described an ecosystem of smart contracts that combined with their corresponding client scripts can perform metering and billing of energy on complex business logic. Converting the billed energy for each department to a fiat currency during the billing process is done outside of the proposed system. We proceed to evaluate the performance of the implementation described in Chapter 5, and explain how the findings from Chapter 3 and Chapter 4 are applied to enable better scalability and robustness in the contract design.

6.1 Scalability

On scalability, we applied Method 3 (3.3.2.3) and followed the rules from Section 3.3.1 to save gas. More specifically:

1. The optimizer flag was activated during compilation

2. Using libraries via Method 3 for storing meter readings and department values resulted in significant code reuse, and reduction of gas costs. Table 6.1 illustrates the gas savings that the proposed method provides, for each function of the contracts.

3. Any meter, department that is no longer used can be deactivated, freeing up storage in the smart contract.

4. Any iterations over arrays are initializing the break condition outside of the loop, reducing the number of `SLOAD` operations done and thus saving gas.

5. In all cases, `string` is avoided and `bytes32` is used.

6. The minimal amount of data needed is stored in the smart contract. The latest reading and timestamp values are needed for each meter and department, and each department additionally requires storing its hierarchical relationship with other departments, virtual meters and its delegates. The formula for each virtual meter (5.3.1) is stored locally because it would significantly increase gas costs to store an arbitrarily long `string` variable in the smart contract.

7. Events are emitted for each action providing an inexpensive way to store the historical data of a meter or department. Emitting an event requires significantly less gas compared to storing a value in contract storage [96]. The

disadvantage is that event data cannot be accessed by another smart contract which is acceptable in this case as historical data is only meant to be accessed by a client and not a smart contract directly.

Table 6.1: Gas costs comparison between Method 1 and Method 3 for storing meter readings and department energy consumptions, optimizer runs = 200

Contract	Method 1	Method 3	Improvement
MeterDB::deployment	793279	637403	20%
AccountingDB::deployment	1691164	1088253	36%
MeterDB::ping	57844	31422	46%
AccountingDB::billPower	82025	28250	65%
AccountingDB::fixedSplit	98126	47260	51%
AccountingDB::headcountSplit	231305	55024	76%

6.2 Security

Having studied and understood past literature as well as attacks on smart contracts, the implementation is developed so that none of the known attacks can be reproduced. In Section 4.2.1 we evaluated that Slither can detect a wide variety of vulnerabilities, and can outperform at most cases the other available security auditing tools. Auditing our smart contracts through Slither showed no true positives which, given the past performance of the tool, gives us a high degree of confidence with regard to the robustness of the developed smart contracts against known attacks.

We utilized and applied an access control list to the implemented smart contracts to enforce role based access control. Roles are allowed to access only the functions of a smart contract that they absolutely must have access to, following the security principle of least privilege. Each user is then granted a role, which provides a portable method to revoke and modify the permissions of a user.

As smart contract ecosystems become more complex and the number of participating parties increase, ways to enforce access control are going to be increasingly important. Our approach allows for a central directory of permissions to exist, which makes it efficient to modify the permissions of each actor on a per-contract basis.

6.3 Complete overview

The final implementation was tested with a setup of 48 smart meters, 44 virtual meters and 20 departments. The simulation was conducted on a Dell Laptop and consumed less than 10% of the laptop's CPU and RAM resources. The deployment of the smart contracts was firstly tested on a local ganache testnet instance, then on a locally deployed testnet of 2 geth instances, where 1 was acting as a node and another as a miner, and finally on the Ropsten public testnet. Due to company requirements, there was no deployment to a public network, however since deployment to Ropsten was successful, it is expected that deployment to the mainnet will be successful as well.

Finally, we analyze the expected operating gas costs for operating the metering and billing logic of the developed smart contracts on the Ethereum mainnet for a year:

1. **Meters:** The monitoring server is updated every 12 minutes, however since the change in values is very small, we poll the monitoring server and update the values of the smart meters every 2 hours[1]. As a result, the operating cost for pinging the values of 48 meters for a year[2] is: 48 meters * 31422 gas * 0.5 readings / hour * 24 hours * 365 days * 1 Gwei/gas = 6.6 ether, which reduces to each meter needing 0.1375 ether in order to be able to interact with the smart contract at the specified frequency for a year.

2. **Virtual Meters:** Pushing the values of Virtual Meters is required to be done twice every month and so the total cost of running 44 virtual meters is: 44 virtual meters * 31422 gas * 2 readings / month * 12 months * 1 Gwei/gas = 0.03 ether, which reduces to each virtual meter needing 0.0007 ether to interact with the smart contract at the specified frequency for a year.

3. **Billing:** Calculating the consumption of each business unit and performing cost forwarding for internal accounting is required to be done once every month. It was simulated that a full billing cycle of pulling values from virtual meters, updating each department's consumption and delegating it according to the business logic costs X ether per month, or X ether per year.

[1] The selected frequency was a company requirement, as more frequent pings would increase cost and provide no significant advantage in granularity

[2] The chosen gas price is $1 GWei = 10^{-9} ether$ since the operations are not time critical. When there is a need for fast confirmations the price can be set higher.

7 Conclusion and Future Work

7.1 Conclusion

In this Diploma Thesis we explained the functionality and applications of blockchain technology. Executing complex operations and transactions at scale can result in significant costs realized as transaction fees and as a result blockchain scalability is considered to be a large issue. A blockchain should be viewed as a tool and not a panacea and should be selectively used where there is need for a replicated append-only database that operates without trust in a third party about the integrity, availability and transparency of data.

Ethereum is a blockchain platform which allows the creation of Decentralized Applications by allowing Turing complete programs to be executed inside its Ethereum Virtual Machine. Ethereum inherits the scalability issues of blockchains. The functionality of smart contracts is limited by the throughput of the network, and it is impossible to perform large amounts of transactions while maintaining reasonably low fees. We claim that solving scalability in Ethereum requires adjustments both at the network level and at the smart contract level. We briefly described the network level scalability solutions and enumerate programming rules and best practices for better contract level scalability, as well as propose a technique for more efficient data storage in smart contracts.

On security, although the attacks on the Ethereum protocol were limited and quickly remediated, smart contracts that are insecure are consistently being developed and exploited in the wild. We went over past smart contract hacks which have led to heists of over 200 million dollars, as well as cases of funds being unspendable due to smart contract code that is no longer able to be run. We evaluated the usage of the tool Slither and found it to be superior to the other available open source auditing tools.

We developed a suite of smart contracts as well as client scripts that perform metering and billing of energy across a complex set of meters and departments based on a predefined accounting model set by the collaborating company, Honda R&D Europe Germany. The system was deployed on a public testnet and as a result is transparent and decentralized. The transactions are publicly inspectable and verifiable by anyone in the network. It is considered to be reasonably secure[1] given that no true positives were given from Slither and that the authoring of the code was made with taking the industry's best practices into mind, from Chapter 4. An access control list was developed in Chapter 5 to only allow specific actors to have access to functions of the smart contracts. Finally, the system's scalability is improved by following the scalability rules and by utilizing the proposed authoring

[1]No system can be fully secure, unless formally verified.

technique for storing data from Chapter 3.

7.2 Future Work

All of the identified research topics for the Diploma Thesis were explored adequately.

Future work includes research on compiler optimization techniques to remove redundant operations and optimize parts of the reading and writing to storage, as a short term contract level scalability solution. Research on network level scalability should move towards Proof of Stake consensus algorithms to remove the need for the energy-consuming Proof of Work, as well as further research should be performed on the implementations of sidechains, state channels and sharding.

On security, smart contract languages must be developed which by default do not allow basic vulnerabilities such as TheDAO, and ones which are formally verifiable, in order to be able to mathematically prove that the code is not going to deviate from its intended results. Auditing tools should be further improved to analyze not only source code, but bytecode as well. Static analysis is not enough to detect vulnerabilities that are unknown, so there must be progress in the area of symbolic analysis tools.

Due to the aforementioned scalability issues, a use-case involving frequent transactions between multiple devices in an IoT scenario, such as our implementation, has large operational costs and it maybe the case that a permissioned blockchain can perform better, or an alternative distributed ledger architecture with high throughput and low transaction costs. A final improvement on our system is making it so that the client script for each meter is able to be run on the meter itself, thus removing the need for the monitoring server, which is a single point of failure.

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_mO09GtSKEKrAsfO7Frgx18pNU/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: Tangerine whistle." https://github.com/ethereum/EIPs/blob/master/EIPS/eip-608.md, 2017.

[76] S. G. Ethan Heilman. Alison Kendler, Aviv Zohar, "Eclipse attacks on bitcoin's peer-to-peer network." Cryptology ePrint Archive, Report 2015/263, 2015. https://eprint.iacr.org/2015/263.

[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] "Parity multisig wallet exploit hits swarm city funds - statement by the swarm city core team." https://press.swarm.city/parity-multisig-wallet-exploit-hits-swarm-city-funds-statement-by-the-swarm-c
1.

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

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

[85] "Eip-999: Restore contract code at 0x863df6bfa4." https://ethereum-magicians.org/t/eip-999-restore-contract-code-at-0x863df6bfa4/130.

[86] "Use our suite of ethereum security tools." https://blog.trailofbits.com/2018/03/23/use-our-suite-of-ethereum-security-tools/.

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

[88] "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/.

[89] "Openzeppelin, a framework to build secure smart contracts on ethereum." https://github.com/OpenZeppelin/openzeppelin-solidity.

[90] "Ethernaut." https://ethernaut.zeppelin.solutions/.

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

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

[93] "Summary of ethereum upgradeable smart contract." https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c.

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

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

[96] "Technical introduction to events and logs in ethereum." https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e.

Appendices

A Transactions and Blocks

Querying a node for transactions:

```
> web3.eth.getTransaction("0
  x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
  ")
{ blockHash: '0
  x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
  ',
  blockNumber: 5284738,
  from: '0xbc7f7a5eb2039a3330665aa22b5d79485896bd11',
  gas: 250000,
  gasPrice: BigNumber { s: 1, e: 10, c: [
      80000000000
    ]
  },
  hash: '0
    x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
    ',
  input: '0
    x0a19b14a00000000000000000000000000000000000000000000000000000000000000000000
    ',
  nonce: 9,
  to: '0x8d12a197cb00d4747a1fe03395095ce2a5cc6819',
  transactionIndex: 0,
  value: BigNumber { s: 1, e: 0, c: [
      0
    ]
  },
  v: '0x1c',
  r: '0
    xe7df9fee346ea3575dd541e120df7b154d46bba2bf859352b29a7f2c6dd5fab3
    ',
  s: '0
    x68d266b140ac468ee328592ca6b4a461edace859d6837358bc3f8eeeb24ade63
    '
}
```

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

Querying a node for block contents:

```
> web3.eth.getBlock(5284738)
{ difficulty: BigNumber { s: 1, e: 15, c: [
      32,
      85319757566868
    ]
  },
  extraData: '0x7869786978697869',
  gasLimit: 7995219,
  gasUsed: 1547361,
  hash: '0
     x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
     ',
  logsBloom: '0
     x0002080000020400020004000000000000001004000000000000800020000000084000800400
     ',
  miner: '0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb',
  mixHash: '0
     x29b6efa55ad0298b0c90f21e9e23d572977ffb3c5064a9816a69bb2bf2a9effd
     ',
  nonce: '0xabed128000fed25e',
  number: 5284738,
  parentHash: '0
     xb7063b9c7b05c95c35a329717e44875829cc740b2e0749e03d54806dcf34b520
     ',
  receiptsRoot: '0
     xe5e176557b9f40394917191095b706a2a331742f0dc93a10e1d59b5e297ee0b5
     ',
  sha3Uncles: '0
     x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
     ',
  size: 7789,
  stateRoot: '0
     x1c62917ac72a2b76e00053efbb7af0d6949e86cafb3f983812d763715c6c9905
     ',
  timestamp: 1521484243,
  totalDifficulty: BigNumber { s: 1, e: 21, c: [
      31406307,
      78318927526632
    ]
  },
  transactions: [ '0
     x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
     ',
     '0
        xbe1c3e767e34d5d668ea50d3400b2e11a663479f931c225eda5e1d314e012589
        ', ...
  ],
  transactionsRoot: '0
     xb0a066469d74fe1f450c5fa8a1f59c5b7305feb6336d0d59f347a2b2c7a8c579
     ',
  uncles: []
}
```

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

Ganache UI:

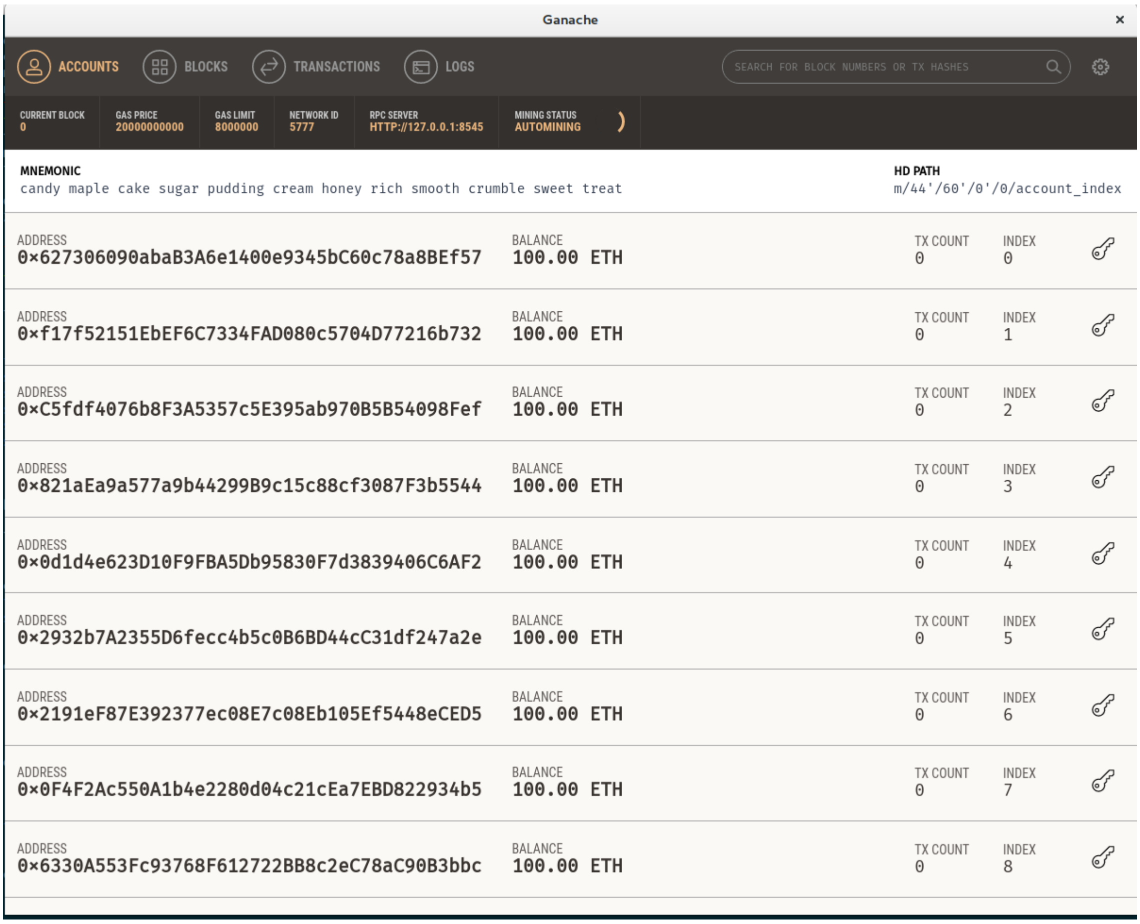


Figure A.3: Ganache testnet User Interface

Web3js example:

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

Figure A.4: Interacting with a node in Javascript

Web3py example:

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

Figure A.5: Interacting with a node in Python

B Scalability through Gas Saving masks

```
pragma solidity ^0.4.21;

library L {
    function add(uint a, uint b) public pure returns(uint) {
      return (a+b);
    }
}

contract C {
    using L for uint;
    uint public x = 1;
    uint public y = 2;

    function add() {
        x = x.add(y);
    }

}
```

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

```
pragma solidity ^0.4.21;

library L {
    function add(uint a, uint b) public pure returns(uint) {
      return (a+b);
    }
}

contract C {
    using L for uint;
    uint public x = 1;
    uint public y = 2;

    function add() {
        x = x.add(y);
    }

}
```

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

```
pragma solidity ^0.4.21;

contract Packing {

    uint64  a;
    uint64  b;
    uint64  c;
    uint64  d;
    uint128 e;
    uint128 f;

    function set() public {
        a = 1;
        b = 2;
        c = 3;
        d = 4;
        e = 5;
        f = 6;
    }
}
```

```
$ solc --optimize --asm Packing.sol | grep sstore | wc-l
2
$ solc --asm Packing.sol | grep sstore | wc -l
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:

```
pragma solidity ^0.4.21;

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

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

}
```

Figure B.4: Interface for described use-case

Tightly packed code:

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

(a) Character structure definition

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

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

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

    Characters.push(c);
}
```

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

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

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

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

Method 2 code:

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

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

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

      Characters.push(c);
  }
```

Figure B.6: Create Character by shifting variables

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

Figure B.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:

```
    function GetProperty(bytes32 Character, uint mask, uint shift)
        private pure returns (uint property) {
         property = mask&(uint(Character)/shift);
    }

    function SetProperty(bytes32 Character, uint mask, uint shift,
        uint value) private pure returns (bytes32 updated) {
         updated = bytes32((~(mask*shift) & uint(Character)) | ((
            value & mask) * shift));
    }
```

(a) Getting and setting a property

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

(b) Mask and shift offsets for CreationTime

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

```
bytes32 c = c.SetPlayerID(
    playerID);
c = c.SetCreationTime(
    creationTime);
c = c.SetClass(class);
c = c.SetRace(race);
c = c.SetStrength(strength);
c = c.SetAgility(agility);
c = c.SetWisdom(wisdom);
c = c.SetMetadata(metadata);
```

(a) Create Character by shifting variables

```
bytes32 c = Characters[index];
return (
     c.GetPlayerID(),
     c.GetCreationTime(),
     c.GetClass(),
     c.GetRace(),
     c.GetStrength(),
     c.GetAgility(),
     c.GetWisdom(),
     c.GetMetadata()
);
```

(b) get character variables

C Security

C.0.1 Deleting a struct with mapping

```

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

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

D Implementation

D.1 Listening for Events

D.2 Access Control

Explain modifiers and whitelist. Compare to having ACL contract.