Blockchain-Based Energy Metering and Billing: Improving Scalability, Security, and Efficiency

Tasmiya Mujawar
mtasmiya44@gmail.com

***Abstract*—This thesis explores the potential of blockchain technology, specifically Ethereum, in addressing inefficiencies in the energy market. The research focuses on enhancing scalability and security of smart contracts, crucial components for decentralized applications (DApps) in the energy sector. Three key research topics are investigated: improving scalability of smart contracts, enhancing smart contract security, and modeling a decentralized energy metering and billing system on the Ethereum blockchain. The findings contribute to the development of efficient and secure smart contracts, with a proof of concept implementation for energy metering and billing using Solidity. The study emphasizes the importance of industry best practices and evaluates automated security testing tools to ensure the robustness of smart contracts in real-world applications.**

I. Introduction

A. Motivation

In 2009 SatoshNakamoto 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 2014, a blockchain platform that 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 do 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 to achieving scalability and high transaction throughput, which traditional centralized payment processors or server architectures provide.

B. Research Topics

I aim to explore the following research topics in this thesis:

1) Blockchain architectures are hard to scale because of the need for decentralization while maintaining security. In particular, Ethereum's smart contracts allow more complex state transitions, which leads us to research ways to improve the scalability of smart contracts.
2) Due to potentially large financial amounts being held in smart contracts and given the precedent of funds being stolen, I research how smart contract security can be improved and how proper auditing can be streamlined in the development process.
3) I explore how can a system that is able to measure and bill the energy consumed by a set of energy meters be modeled through the use of the Ethereum blockchain. 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 of transactions. Finally, the system needs to be scalable at a reasonable cost.

C. Scope

I examine important quality aspects of DApps like scalability and security and apply the findings in a proof of concept of energy metering and billing using Solidity smart contracts.

On security, I limit ourselves to the industry's best practices, as per the literature and evaluate the performance of an automated security testing tool compared to other tools.

The architecture of the metering and billing model of energy consumed by the set of energy meters mentioned in Section I-B is implemented based on a specific use-case and technical requirements set by Honda Research and Development Europe Germany, hereafter referred to as HREG, with which I collaborate for the purpose of the Thesis.

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

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

D. Outline

Chapter II introduces the terminology required for understanding blockchain fundamentals. I 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 VII, I 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. I make a contribution to scaling smart contracts, which allows for more optimized writes to the Ethereum blockchain.

In Chapter XI, I go over the most significant security issues found in Ethereum and its smart contracts. Having understood these, I evaluate and augment the auditing tool *Slither*'s ability to detect and identify vulnerabilities and compare it to the taxonomy of tools described in [5].

In Chapter XIV, I present ways in which smart contracts and supporting software architectures can address the energy market's inefficiencies and create a suite of smart contracts which are able to answer research question 3 from I-B, while taking the lessons learned from Chapter VII and Chapter XI into account.

In Chapter XX, I evaluate the performance and the extent at which the smart contracts from Chapter XIV achieve their goal.

In Chapter XXIV, I reiterate and summarize the findings from the previous chapters and highlight future work that can be done to further improve both the design of the described smart contracts and the security and scalability of Ethereum.

II. Ethereum and Blockchain Basics

A. General Background

Before getting into the specifics of blockchains and Ethereum, this section will be used to explain fundamental terms of 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 [2]. 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]A transaction can be either a transfer of value between multiple parties or in more complex cases a call to an executable function of a smart contract

B. 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[3] [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.

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].

C. 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*[4]. 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 that 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[5] 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 the sender's private key, the only valid decryption can be done with the

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

[4]The public key is a number that 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.

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

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:
   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.

D. 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[6], 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 canceled due to a court order). On the other hand, in the case of private key theft, there is no way to prevent an attacker from stealing a victim's funds. At the 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 and for scenarios such as reverting the bank balance of a customer in the case of a mistaken transaction.

E. Blockchain Types

In order to tackle the disadvantages mentioned in Section II-D, blockchains that make tradeoffs in decentralization for more transaction throughput or privacy exist.

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, but immutability is not absolute.

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

III. 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*.

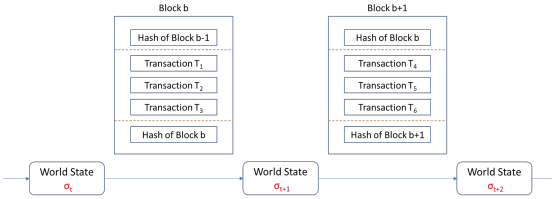


Fig. 1: Ethereum can be seen as a chain of states [23]

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

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[7] 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 as 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[8]. 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[9] 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*[10]. 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.

IV. 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, I go over the internals of the EVM.

A. Accounts

World State: Ethereum's world state is a mapping between the addresses of accounts and their states. Full nodes download the blockchain, execute and verify the full result of every transaction since the genesis block. Users should run a full node if they need to execute every transaction in the blockchain or if they need to swiftly query historical data.

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[11].

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 5

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.

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.

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[12].

I describe the contents of the Account State shown in Figure 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'[13].
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 the empty string (''), while for CAs, it is the KECCAK-256 of the bytecode that exists at the CAs address.

B. Transactions

A transaction is a specially formatted data structure that gets signed by an EOA[14] and gets broadcasted to an Ethereum node. A transaction in Ethereum specifically contains the following:

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[15].
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 IV-D).
5) **gasLimit:** The amount paid by the sender per unit of gas.
6) **hash:** The transaction hash.

[7]The Ethereum network's native currency

[8]This depends on block propagation time based on bandwidth, block-size, connectivity, etc.

[9]Difficulty is a measure of how much computational effort needs to be performed by a miner to solve a PoW puzzle. Total Difficulty is the sum of the difficulties of all blocks until the examined block.

[10]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

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

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

[13]1 ether $= 10^{18}$ we

[14]With the EOAs private key

[15]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*)

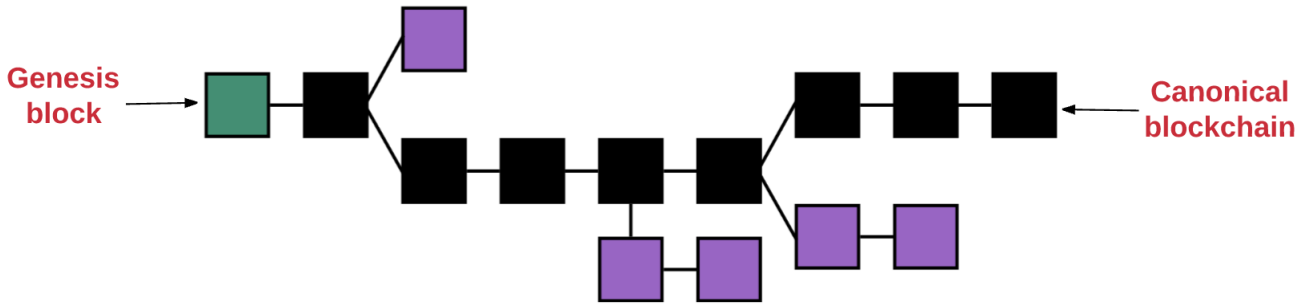


Fig. 2: The Ethereum protocol chooses the longest chain [27]

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

C. Blocks

A block contains the block header and a list of transaction hashes for all of the included transactions in that block.

Specifically, a block in Ethereum contains:

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 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.

D. 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 be completed 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 involve 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 block gasLimit and block size in Bitcoin in that both are used to limit the number 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 weis calculated from the formula

$$totalTransactionCost = gasPrice * gasUsed \quad (1)$$

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 times of network congestion such as popular Initial Coin Offerings[16][30] or mass-driven games such as CryptoKitties[17][31]

[16]Crowdfunding for cryptocurrency projects which allow investors to buy tokens in a platform

[17]https://cryptokitties.co

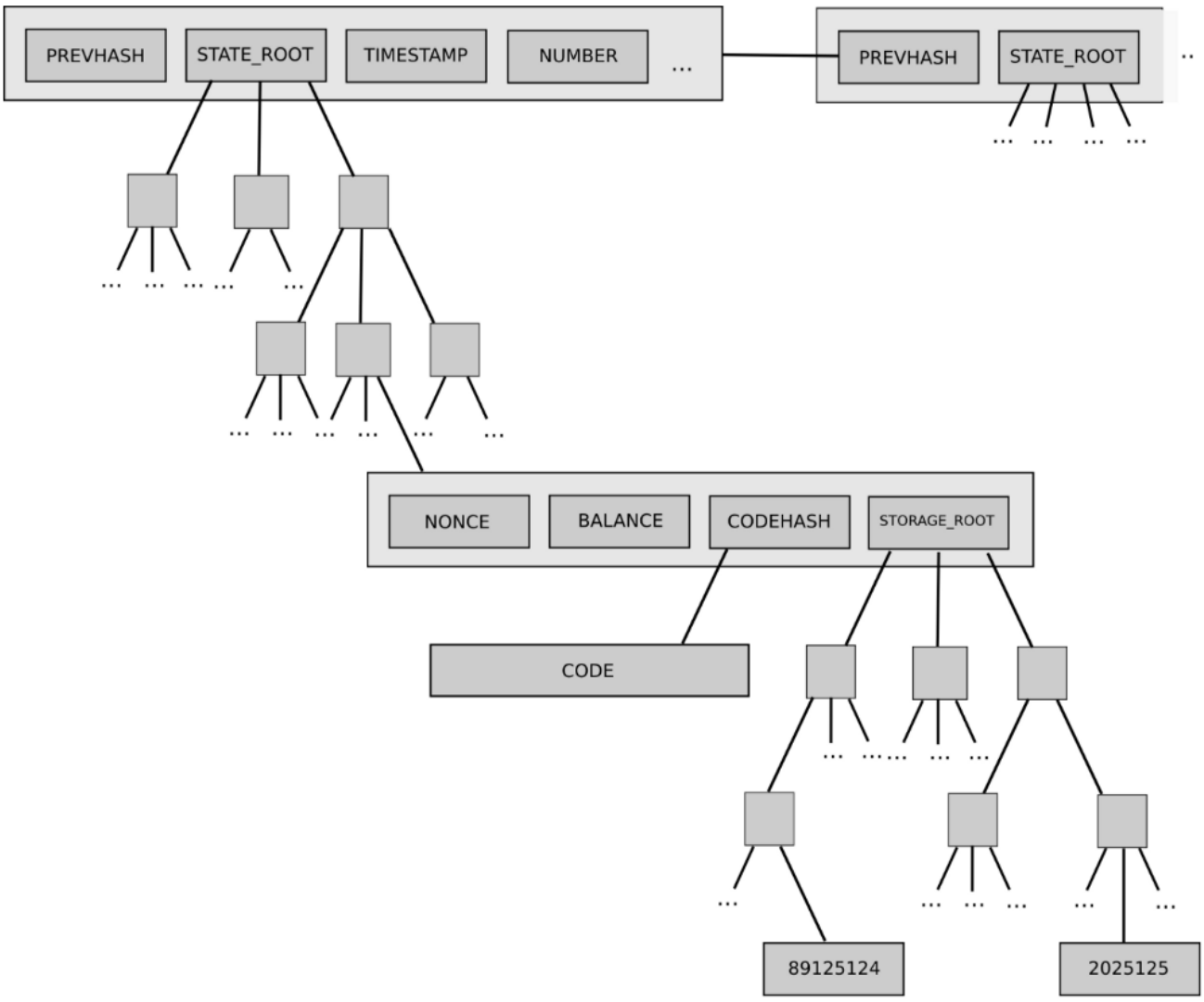


Fig. 3: The Ethereum world state [3]

transactions become very expensive and can take long times to confirm.

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.

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 [18].

E. Mining

The set of rules that allow an actor to add a valid block to the blockchain is called a *consensus algorithm*. In order to have consensus in distributed systems, all participating nodes must have the same version (often called history) of the system. If there were no rules for block creation, a malicious node would be able to consistently censor transactions or double-spend [32]. In order to avoid that, consensus algorithms elect a network participant to decide on the contents of the next block.

Ethereum uses a consensus algorithm called ethash[9], which is a memory-hard[19] 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 so that the output number is less than a certain

[18]https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql

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

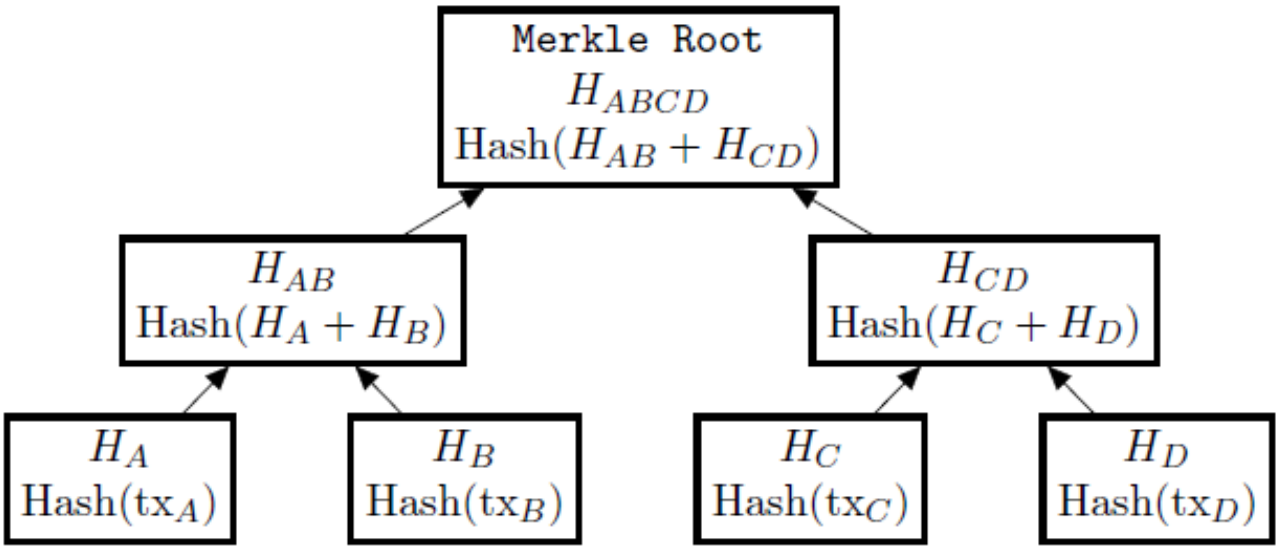
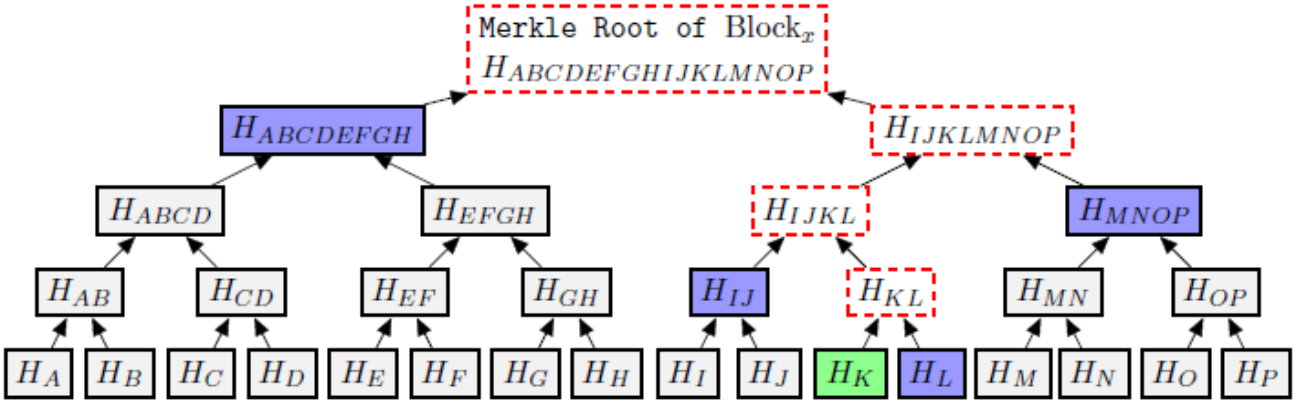


Fig. 4: Node calculation in a Merkle Tree [28]



:

Fig. 5: Merkle path calculation [28]

threshold[20]. 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 VII.

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

V. 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.

A. Programming Languages

Languages that are compiled into EVM code are Solidity, Serpent, LLL, and Vyper. Active research is being done towards more safe and secure smart contract languages [34], [35]. Solidity is the most popular language in the ecosystem, and although often comparable to Javascript, I 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

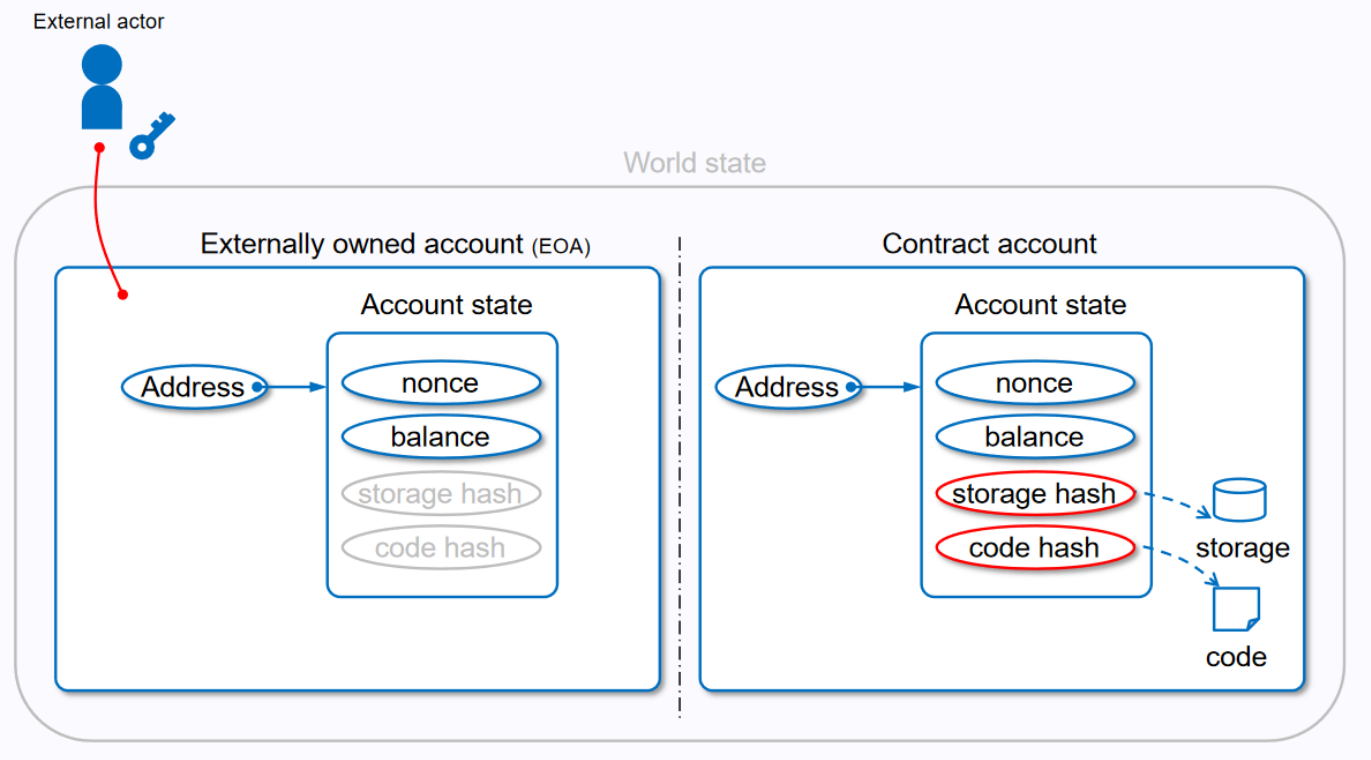


Fig. 6: Externally Owned Accounts and Contract Accounts in the Ethereum World state [23]

and its Application Binary Interface (ABI) are needed, which can be obtained from the compiler.

B. Tooling

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

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 Ethereum (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 [36] consensus algorithm, while Ropsten is running Ethash [9] with less difficulty.

I provide a listing of the currently public 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[21](User Interface at 23, formerly known as testrpc).

2) Web3: Web3 is the library used for interacting with an Ethereum node. The most feature-rich implementation is Web3.js[22] 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[23]. I illustrate an example of connecting and fetching the latest block from Ropsten and Mainnet using Web3.js in 27 and Web3.py in 28. The full specifications of each library's APcan be found in their respective documentation[24][25]

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

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

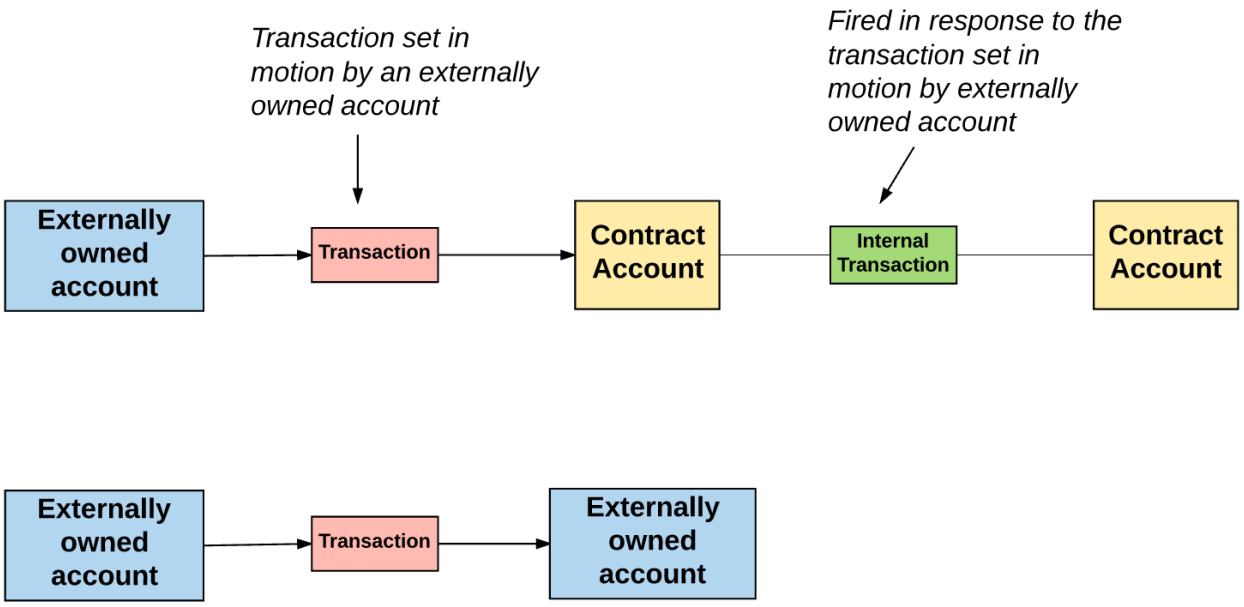


Fig. 7: Transaction made by an EOA to an EOA or a contract [27]

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.

VI. Related Work

Techniques which illustrate more efficient smart contracts by storing less data on a blockchain are described in [37]. With this method, it is possible to store the data in the input of a transaction, removing the need of smart contract. The data can then be processed in an application by watching for transactions that get sent to the contract's address and decoding their inputs.

On network level scalability there are various approaches such as executing transactions 'off-chain'[26] and use a blockchain only for the final settling of a series of transactions by utilizing a technique called payment channels [38], [39], [40], [41]. Other approaches exist which involve creating 'sidechains' which can be utilized to perform trustless computation on another blockchain, that is pegged to the security of the root chain [42], [43], [44], [45], [46]. Finally, another approach to achieving scalability is via permissioned blockchains which trade decentralization and transparency for efficiency [21], [47].

Cheng et al make it clear that compiler optimizations in smart contracts still need improvements in order to avoid unnecessary expenses [48]. They identify 7 gas-costly patterns and develop a tool called GASPER that is able to identify 3 of these patterns from the bytecode of already deployed smart contracts. The evaluation of the frequency of these patterns show that over 70% of the deployed smart contracts on Ethereum until Nov. 5th. 2016 suffer from these 3 patterns. Although experienced software engineers can avoid these bad practices, the compiler should be able to detect and optimize parts of the code which are unreachable, and not perform redundant operations.

Extensive analyses have been performed on the security of blockchains as networks and specifically on the security of Ethereum smart contracts [26], [49]. While the Ethereum network as a Proof of Work blockchain has not been successfully attacked since its inception, Smart contracts have proven to be insufficiently secure for the amounts of funds that they hold, with large amounts of funds being stolen in multiple occurences [50], [51], [52]. As a result, tools that are able to analyze both source code and compiled bytecode for vulnerabilities have been developed [53], [54], [55], [56], [57], [58]. A recent study identifies 'greedy', 'prodigal' and 'suicidal' smart contracts that can freeze or cause loss of funds, however there has been criticism on the validity of the authors' findings [59], [60].

Utilizing blockchain for Internet of Things is explored in [61], [62], while a model for billing and accounting with smart contracts is proposed in [63]. In all cases, it is acknowledged that existing blockchain infrastructure cannot support the amount of transaction load that is needed by the IoT ecosystem. Energy market use-cases are being piloted by introducing peer to peer energy market trading, while utilizing hardware agents to facilitate the energy management [64], [65]. Finally, microgrid blockchain prototypes are being tested such as [66], [67].

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

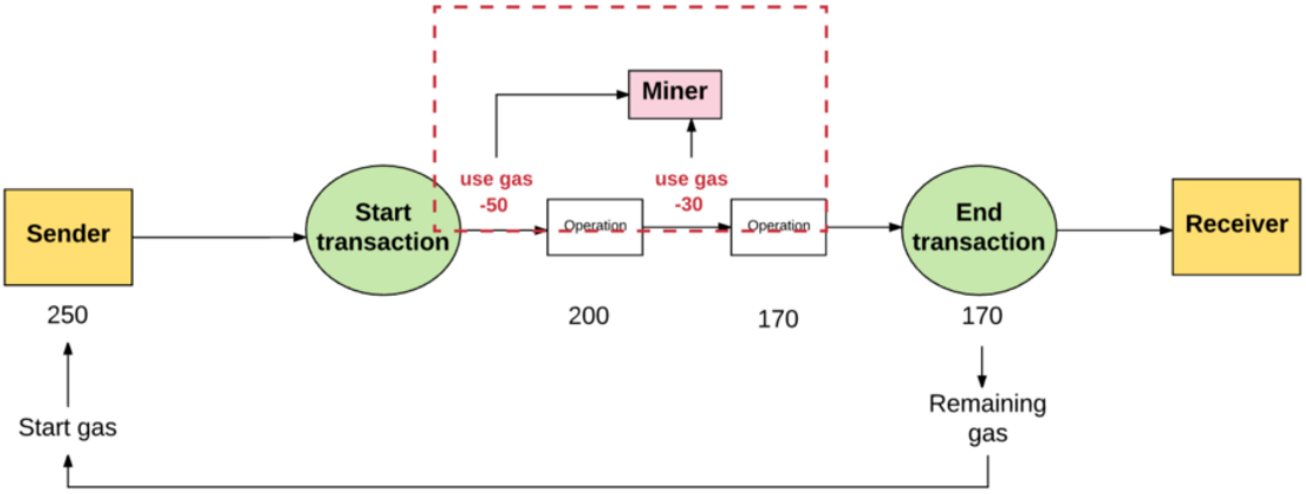


Fig. 8: Successful transaction [27]

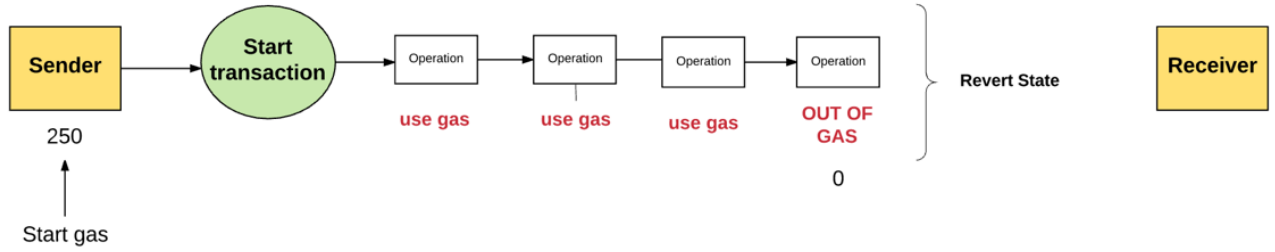


Fig. 9: Failed transaction that ran out of gas [27]

VII. Blockchain Scalability

VIII. 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 seconds on average, and can contain only a finite amount of transactions [68]. As a result, transaction throughput is bound by the frequency of new blocks and by the number of transactions in them.

I 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[27], each block would be able to verify 381 transactions, or 25 transactions per second (tps), which is still not comparable to traditional payment operators.

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

IX. 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. This is described as the *Scalability Trilemma* [69].

> Scalability Trilemma, Sharding FAQ **This sounds like there's some kind of scalability trilemma at play. What is this trilemma and can I break through it?**

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

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

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, more 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 [70], Proof of Work is a consensus algorithm optimized for censorship resistance and high security in transactions. Bitcoin and Ethereum's PoW networks have slow probabilistic time to finality and do not scale well. PoW is supposed to have a low barrier to entry since anyone is able to buy mining equipment, but due to economies of scale, PoW blockchains end up being centralized around small numbers of miners [71]. As a result, Bitcoin and Ethereum belong to the lower right angle of the Scalability Triangle, as shown in Figure 10[28].

I 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 place of miners, there are validators who, instead of expending computational resources to 'mine' a valid block, they stake[29] their ether and the probability for them to be elected to validate the next block is proportional to their stake. Proof of Stake can reduce the energy waste caused by mining, as well as provide faster block times and thus increase the transaction throughput of a secure and decentralized blockchain. Designing a secure PoS protocol is still under heavy research. The Ethereum Foundation is working on 'Casper the Friendly Finality Gadget' [72], which is a hybrid PoW/PoS consensus algorithm that provides block finality[30] which combined with the 'correct-by-construction Casper the Friendly GHOST'[31] [73] will enable a full transition to Proof of Stake.

[28]DBP means Decentralization of Block Producers

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

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

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

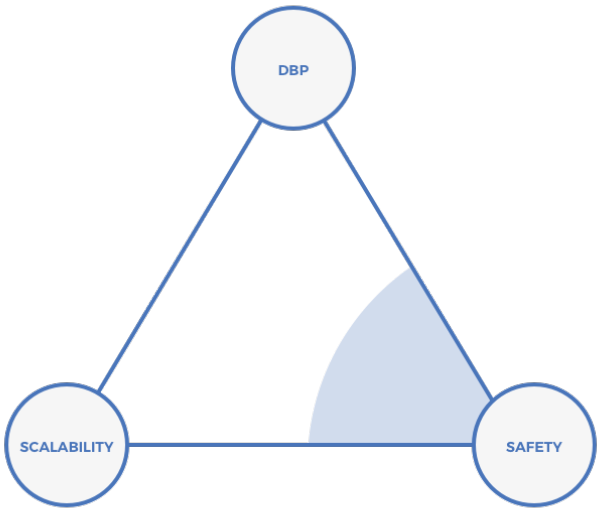


Fig. 10: The Scalability Triangle [70]

Sidechains: A sidechain [42] is a blockchain defined by a custom 'rule-set' which can 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' [46] and involves creating *Plasma chains* that run their own consensus algorithm and communicate with the mainchain via a two-way peg as described in [42]. *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' [45] 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 [69] refers to splitting the process across nodes so that each full node is responsible only for a shard[32] and acts as a light client to the other shards. Sharding is the most complex scaling solution and is still in the 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 involve the 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 performing a

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

transaction on the blockchain only when the process is done[33].

X. Contract Level Scalability

In a recent study, after evaluating 4240 smart contracts, it is found that over 70% of them cost cost more gas than they should [48]. This happens because the compiler is unable to optimize away the identified gas-costly patterns. In this case, experienced software engineers are expected to be able to identify and avoid them on their own.

It also turns out that the creation of `struct` type variables in Solidity is heavily underoptimized, with the Solidity compiler performing a large number of redundant operations [74]. In this section, I explore how gas gets computed in smart contracts and potential ways I can save gas and, as a result, transaction costs.

A. 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 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 I: Gas costs for different operations [48]

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 I, the most expensive operations involve CREATE, CALL, and SSTORE. The focus of this section will be to explore ways to decrease gas costs on Smart Contracts, either through better storage practices or by handcrafting optimizations for specific use cases. I choose to optimize the usage of SSTORE because it gets executed whenever a storage variable is modified, contrary to CREATE and CALL, which get executed when a contract creates another contract or makes external calls to another contract.

It should be noted that non-standard methods have been proposed to reduce fees incurred by gas costs. A recent construction [75] describes a method of buying gas at low-cost periods and saving it in order to spend it when gas prices are higher[34]. 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 led to unexpected scenarios [76]).
2) Reuse code through libraries [77].
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 [48].
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[35].

As described in [48] there is a lot of room for further compiler optimizations. Future Solidity compiler versions are addressing some already[36][37][38].

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 F for an example of the optimizer's functionality.

B. Gas Savings Case Study

I 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.

[33]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.

[34]When the network is congested

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

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

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

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

TABLE II: Required variables and sizes 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 II 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.

I 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 [78].

I 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 F. A comparison of the performance for each method is shown in X-C

1) Packing of traits by utilizing Solidity's optimizer and `struct` variables: I use Solidity's built-in `struct`[39] keyword as means to group all traits of a `Character` as described in II. This allows for easy code readability since every variable of a `struct` can be accessed by its name 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
);
```

Fig. 11: Method 1 API

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, I can create an array of `Character` type structures and access their traits by their indexes. The gas costs per function call with this method are shown in III.

TABLE III: 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

2) Manually pack traits in a uint256 variable with masking and shifting: In X-B1 I rely on the optimizer to make storing a character's traits more efficient. In order to get better results, I create a local stack variable[40] which is large enough to store all the traits from II. Instead of creating a `struct`, I manually encode each trait in the said variable, essentially I 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. I 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)
);
```

Fig. 12: 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 13 14.

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.

The gas costs per function call with this method are:

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

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

[40]The data is 248 bits long, so I create a `uint256` variable

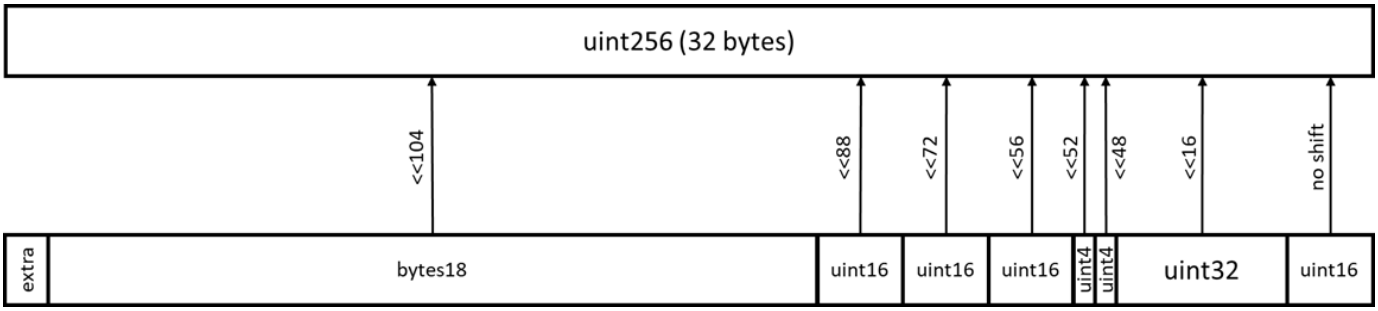


Fig. 13: Encoding data in a `uint256` variable

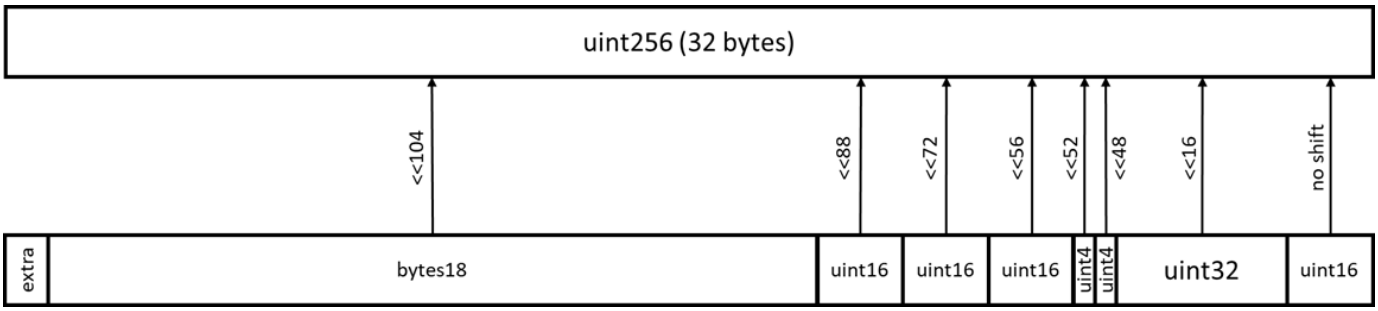


Fig. 14: Encoding data in a `uint256` variable

TABLE IV: 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

packing is very efficient in X-B2 compared to X-B1, the code is hardly readable and it is impossible to reuse parts of it. I 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[41]. A simple example is shown in 30.

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.
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 31. They often include generic functionality such as math operations[42]. Depending on the complexity of the contracts, this can be more efficient compared to using `internal` functions.

The final version of Method 3 is split in two files, the library which includes the APfor 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 I are able to store and retrieve a character's traits in a user-friendly manner, as shown in Figure 15.

```
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()
);
```

Fig. 15: 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. I use `internal` because it requires less gas

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

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

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 V: 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

C. Results

Observing III, IV and V, 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 either for size or for runtime costs [79]. In this section I compare the gas costs for calling the `CreateCharacter` function and for deploying the contract for 1 optimizer run[43].

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 12. 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, which is considered a potential security threat.

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 11) with Method 3 (Figure 15) it is seen that they have very similar syntax, with Method 3 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, I evaluate on a weighted score based on *Readability*, *Portability*, *Gas Efficiency* and *Security*, as illustrated in Table VI. The values of the weights are chosen based on the observations made on smart contract optimization with respect to security. Compared to Method 2, Method 3's score higher on portability because it utilizes Libraries which by definition are more portable, and it also scores higher on security due to the ability to reuse code which has been audited and tested.

The results of the evaluation are shown in Table VII, on a scale of 0 to 5, with 5 being the highest value. Based on that, I choose to use Method 3 in order to make the implementation described in Chapter XIV more gas efficient while retaining its readability and maintainability.

XI. 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[44]. In this chapter I 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. It should be noted that the security of a smart contract is a completely different problem set from the security of the network. I 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. I also improved *Slither* by augmenting the scope of vulnerabilities it was able to detect.

XII. Past Attacks

A. Network Level Attacks

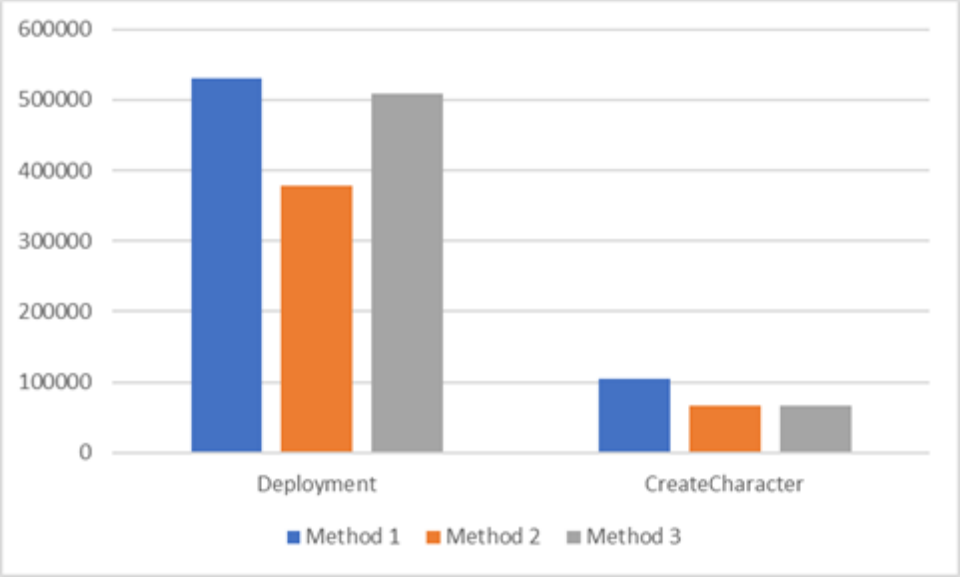
a) October 2016 Spam Attacks: During the period of September-October 2016, an attacker was able to flood the Ethereum network's state by creating 19 million 'dead' accounts. The attack was made possible by a mispricing in the SUICIDE opcode of smart contracts, allowing an attacker to submit transactions 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 [80]. As a response, two hard-forks[45] were proposed. The first one, Tangerine Whistle[81] solved the gas pricing issue and at a later point, Spurious Dragon[81] cleared the world state from the accounts created by the attack.

b) 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 [82]. On the other hand, Eclipse attacks on Ethereum require only a

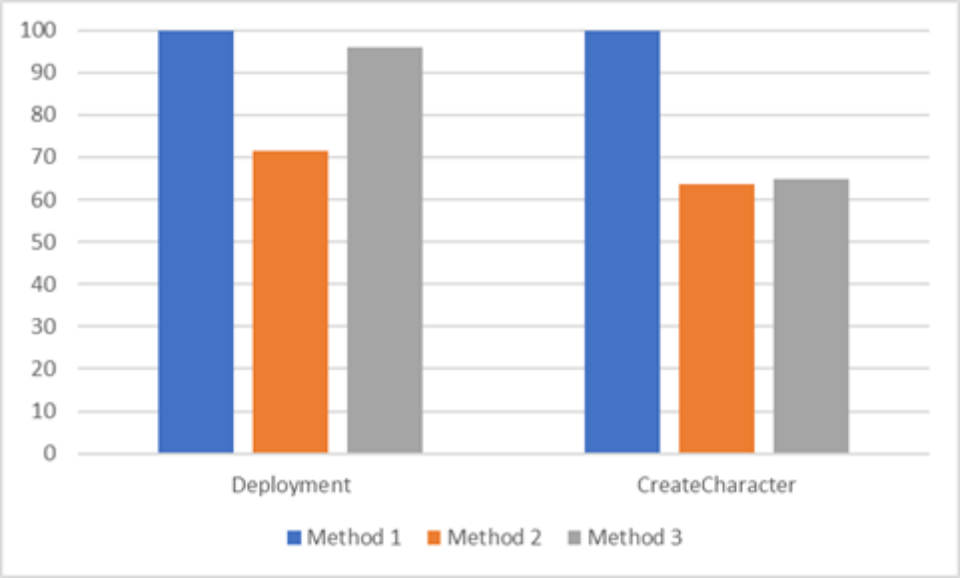
[43] I do not compare `Register` because the implementation is the same across all 3 methods and `GetCharacterStats` does not cost gas to call externally because it is a `view` method which does not modify the state.

[44] Double-spending is the act of successfully spending money more than once

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



(a) Absolute values in gas for each method



(b) Percentage comparison between each method

Fig. 16: Gas cost comparison between the 3 proposed methods.

small number of attacking nodes, because it uses a structured network protocol contrary to Bitcoin's unstructured network protocol which makes random connections [83]. The attack vector was communicated to the client developers and the vulnerabilities were fixed in *geth* v1.8[46]. 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.

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

B. Smart Contract Attacks

Contrary to Ethereum as a network, smart contracts have been exploited multiple times the past. I proceed to give a brief explanation of the three biggest hacks by financial loss in Ethereum smart contracts, the 'TheDAO' and the 'Parity Multisig 1 and 2'[47] which involved two independent incidents with the same smart contract.

[47]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/

TABLE VI: Selected weights for evaluation criteria.

Name	Weight	Description
Readability	0.25	How readable is the code
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?

TABLE VII: 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

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 [84].

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*[48] attack which allowed an attacker to make unlimited withdrawals from the contract [52].

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 [84]. 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' [85].

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 [50]. The attack involved a library contract, which contrary to using Solidity's `Library` pattern discussed in Section X-B3, 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 [86].

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[49]. 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 [87]. 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.

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[50]. 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 [51].

A number of proposals were made in order to recover the locked funds [88], [89]. 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.

XIII. 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

[48]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.

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

[50]https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

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 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 [90], [49], [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[51], however the Solidity Compiler was later updated to throw a warning if this issue exists.

A. 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, I came in contact with TrailOfBits, a security auditing firm, and used their suite of tools to extend the already built taxonomies [91]. I utilized the tool Slither to audit smart contracts. I 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 [53], [54]. I 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[92]. 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 [49].

Reentrancy bugs: Due to the TheDAO incident (XII-B1) a lot of attention was raised on reentrancy 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[52]. 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 K

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, explained in Appendix L.

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[53], using 'emit' for events is going to be mandatory from Solidity 0.5.0 and onwards.

B. Towards more secure smart contracts

Developers must be familiar with the security best practices[54], 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 [93], [94], [95]. Finally, there are techniques which allow a contract to be upgradeable, and thus allowing it to be 'patched' in case a vulnerability is foound. It should be taken into account that while the upgradeability feature can be useful in this case, there is increased cost, complexity and gas costs, as well as may introduce additional attack vectors.

XIV. Metering and Billing of Energy on Ethereum

Smart contracts can transform the energy industry. In this chapter I explore the inefficiencies of the energy market and identify gaps which can be filled by blockchain. I go through the advantages of an energy-based application built on smart contracts. Finally, I describe the business logic of a specific

[51] https://github.com/ethereum/solidity/issues/2628

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

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

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

TABLE VIII: Slither is able to detect more vulnerabilities compared to other tools. Updated table from [5]

Vulnerability	Oyente	Remix	Securify	SmartCheck	Mythril	Slither
Reentrancy	Y	Y	Y	Y	Y	Y
Time-Order Dependency	Y	Y	Y	Y	Y	N
High Gas	N	Y	N	Y	Y	N
Unprotected Function	N	N	N	N	Y	Y
No Constructor	Y	N	N	N	Y	Y
tx.origin	N	Y	Y	Y	Y	Y
blockhash	N	Y	N	N	N	Y
Variable Shadowing	N	N	N	N	N	Y
Mapping in struct	N	N	N	N	N	Y

energy use-case which I implemented on Ethereum. The implementation takes into account the methods and concepts described in Chapter VII and Chapter XI in order to ensure that the smart contracts are efficient and robust.

XV. Energy Market inefficiencies

The global energy market is gradually transitioning to clean and renewable energy. Regulations encourage the usage of distributed energy resources (DERs) [96]. 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 [97]:

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.

XVI. Energy Market use-cases for Blockchain

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

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 XVII).
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.

XVII. Business Logic

In collaboration with Honda R&D Germany, I 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.

I proceed to discuss the business logic of the use-case and then implement it. I utilize the Method 3 from Section X-B3 to optimize my smart contracts for gas efficiency, and utilize Slither from Section XIII-A and the learned best-practices to ensure that the developed smart contracts are robust.

A. 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*[55] its reading directly to the blockchain, I 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 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 for calculating the consumption of a virtual meter can be seen in Equation 2. VM is used for Virtual Meters, ELT for Smart Meters and F for constant factors.

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

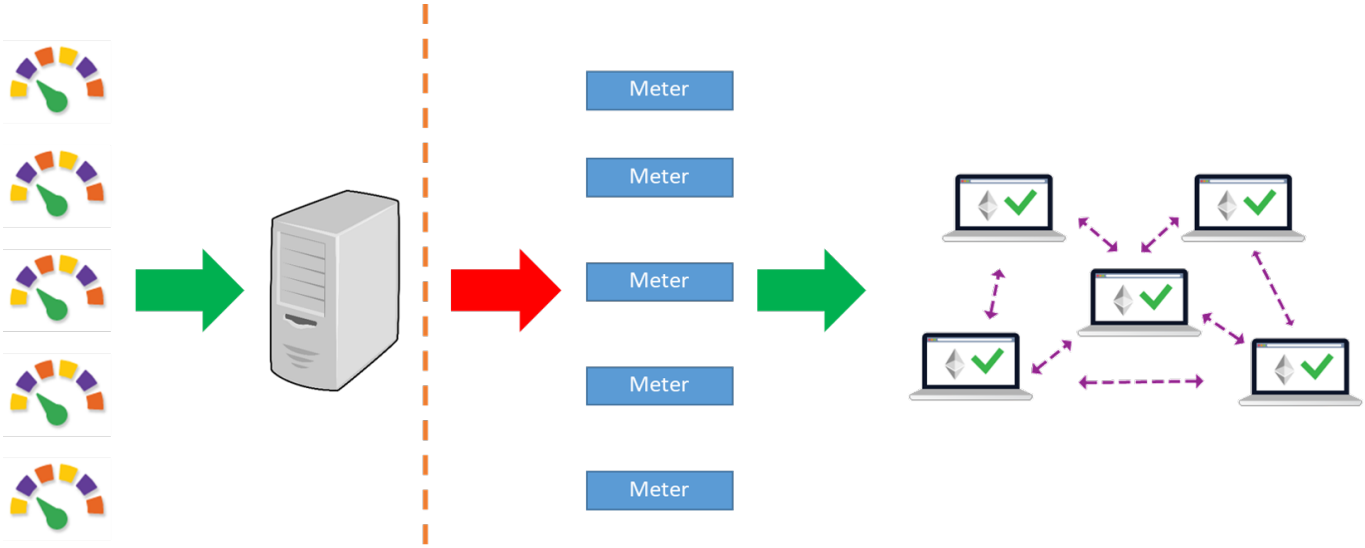


Fig. 17: Meter Data gets pushed to the blockchain after getting pulled from the monitoring server

$$VM01 = 2 * ELT01 + ELT02 + F1 * ELT03 \quad (2)$$

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 18.

B. 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 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 $(100-X)\%$.

As an example, in 19, 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.

XVIII. Smart Contracts

In this section I go over the implementation and the rationale of each developed smart contract. I explain the inner workings and provide tests of their functionality. I 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 P). 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.

A. MeterDB

MeterDB utilizes Method 3 (X-B3) to keep track of the energy consumption values and logged timestamps of a smart meter or a virtual meter. I assume that a meter can be any IoT device that has its own Ethereum account[56]. 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

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

Fig. 18: The consumption of Virtual Meters gets calculated from the consumption of their associated Smart Meters.

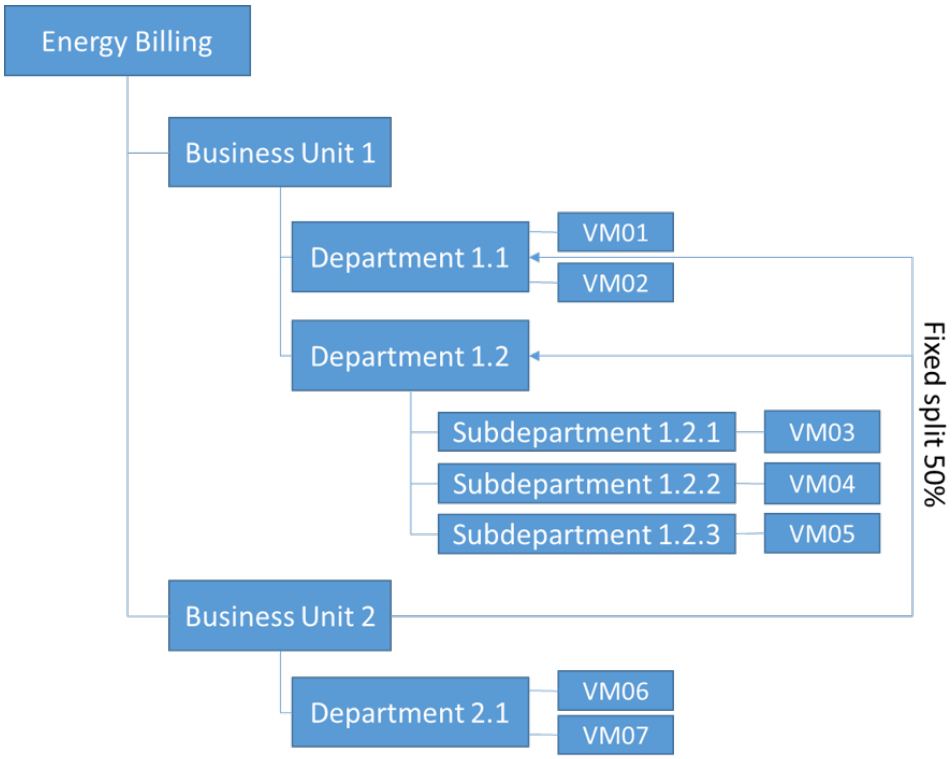


Fig. 19: Relationship between Departments and Delegates.

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 through the `activateMeter` function. Deactivating a decommissioned meter results in its data being deleted from the storage of the smart contract, which refunds gas as explained in Chapter VII. This is done through `deactivateMeter`. I 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 IX 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 milliwatthmy granularity.

B. AccountingDB

AccountingDB stores the data required to performing metering and billing for a department. Each department's data is bro-

TABLE IX: 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

ken down in department data and its accounting data. Storing department data involves using Method 3 (X-B3) to store the consumption of each department, the headcount and activity status of the department and is done in `Department.sol`.

Due to the forwarding of a department consumption to another, I 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 XVII-B.

Departments are activated through `activateDepartment` and deactivated through `deactivateDepartment` and the headcount is set through `setHeadcount`. For each department, the list of Virtual Meters, Subdepartments and Delegates is stored in a `struct`, via the methods `setDelegates`, `setSubdepartments`, and `setVirtualMeters`. Finally, the billing of energy for a department is done through `billPower`, clearing is done through `clearDepartment` and forwarding a department's consumption is done through `headcountSplit` and `fixedSplit`.

C. 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 XIX to reduce the number of addresses that need to be supplied to interact with the smart contracts[57].

D. Access Control

As discussed in Chapter XI 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 O).

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

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`.

E. Deployment Pipeline

The full deployment pipeline involves deploying each contract separately and linking it to the Access Control List as shown in Figure 20. 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.

XIX. Client Side

As described in Section IV, smart contract functions get executed only in response to transactions made by EOAs[58]. Since the goal is to have an automated process of pulling data from and pushing data to the smart contracts, I implement client scripts which perform that functionality. I 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 APof the monitoring server. A separate keypair is generated for every role mentioned in Section XVIII-D, and each CLis designed to function only with by using the corresponding role's private key.

A. Utilities

Web3.py and Contract Wrappers: I 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

[57]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

[58]Externally Owned Accounts, as described in Section IV

TABLE X: 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

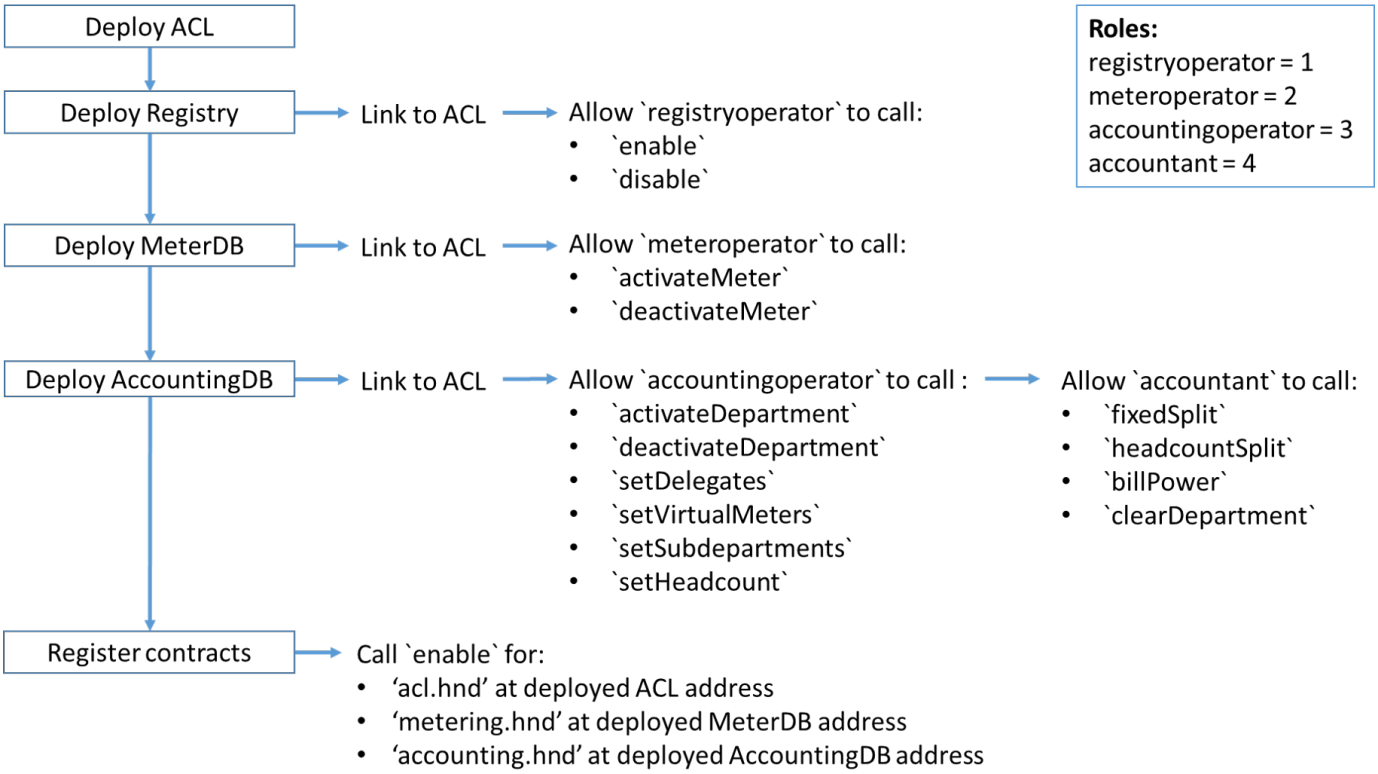


Fig. 20: The full deployment pipeline enforcing access control

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.

I 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 ABand 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. I use the `Contract` class throughout my 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 XVII-A, there is no direct access to each meter's reading, which forces us to access all readings through a REST API. I 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 APwas considered out of scope becaues it did not provide any additional information for my use-case.

B. Administrator CLI

The administrator role is used to grant or revoke role permissions to operators. The CLinteracts with the ACL smart contract which has been initialized as described in Section XVIII-D. 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.

C. 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 CLalso provides the functionality to resolve a contract name to its address or a contract address to its name.

D. 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 CLexpects 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 CLtries 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`.

E. 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.

F. Single and Multiple Meters CLI

A `meter` module is implemented which is responsible for connecting to `MeterDB` and the monitoring server APas explained in Section XIX-A. The meter first needs to be activated by the meter operator as explained in Section XIX-D. The client is responsible for polling the monitoring server and ping any new power readings to the smart contract. The `single-meter` CLexpects the meter's private key as input, and retrieves the meter id from the smart contract in order to connect to the monitoring server. A frequency parameter can also be supplied to set how often a meter will poll the monitoring server for new data. This process runs in an infinite loop, or it can be modified to run as a cronjob.

Due to having multiple smart meters in my use-case, a `multiple-meters` CLis 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.

G. 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 XVII-A. Along with the virtual meter's private key, a function is supplied to the CLas 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[59] before pinging the final value to the smart contract. The process runs in an infinite loop, and can be modified to run as a cronjob.

Similarly to normal smart meters, a `multiple-virtualmeters` CLis 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.

H. 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[60]

The departments are processed in order from the innermost tiers towards the top level departments, as described in Section XVII-B. The flowchart in Figure 21 shows the process that each department goes through in order to correctly calculate its final consumption.

I. Execution Pipeline

Having described how each module of the client side works, in Figure 22 I 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 time the accountant is able to bill each department according to the logic which was previously specified by the accounting operator.

XX. Evaluation of Implementation and Results

I have described an ecosystem of smart contracts that combined with their corresponding client scripts can perform metering and billing of energy with complex business logic. I proceed to evaluate the performance of the implementation described in Chapter XIV, and explain how the findings from Chapter VII and Chapter XI are applied to enable better scalability and robustness in the contract design.

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

[60] 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.

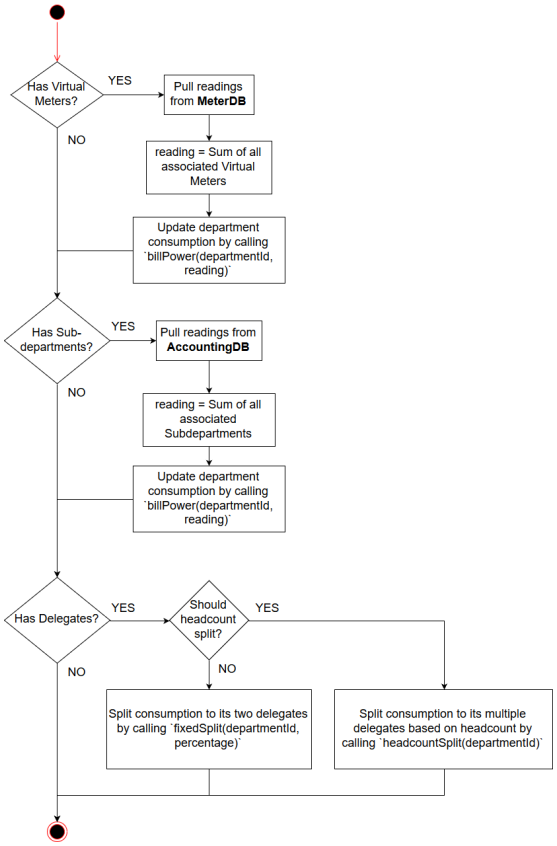


Fig. 21: Consumption calculation for a Department

XXI. Scalability

On scalability, I applied Method 3 (X-B3) and followed the rules from Section X-A to save gas. Method 2 was not evaluated on the smart contracts as it would require significant refactoring, and based on the results from Chapter VII it is expected to have similar performance. 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 XI 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 (XVII-A) 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 [98]. The disadvantage is that event data cannot be accessed by another smart contract which is acceptable in this case as historical data are only meant to be accessed directly by a client and not by a smart contract.

TABLE XI: 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%

XXII. 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 XIII-A I evaluated that Slither can detect a wide variety of vulnerabilities, and can outperform at most cases the other available security auditing tools. Given that the written code does not have

1) Reentrant functions since there is no external call to an address controlled by an attacker.
2) Functions dependent on block inclusion time.
3) block.hash
4) tx.origin
5) Mappings inside structs.

the only vulnerabilities to look for are 'Variable Shadowing' and 'No Constructor'. The only tool able to find both of the issues is Slither, which gave no positive results for the written contracts. Mythril was also executed in order to detect 'No Constructor', which also gave negative results. These results, give us confidence regarding the robustness of the developed smart contracts against known attacks.

I 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. my 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.

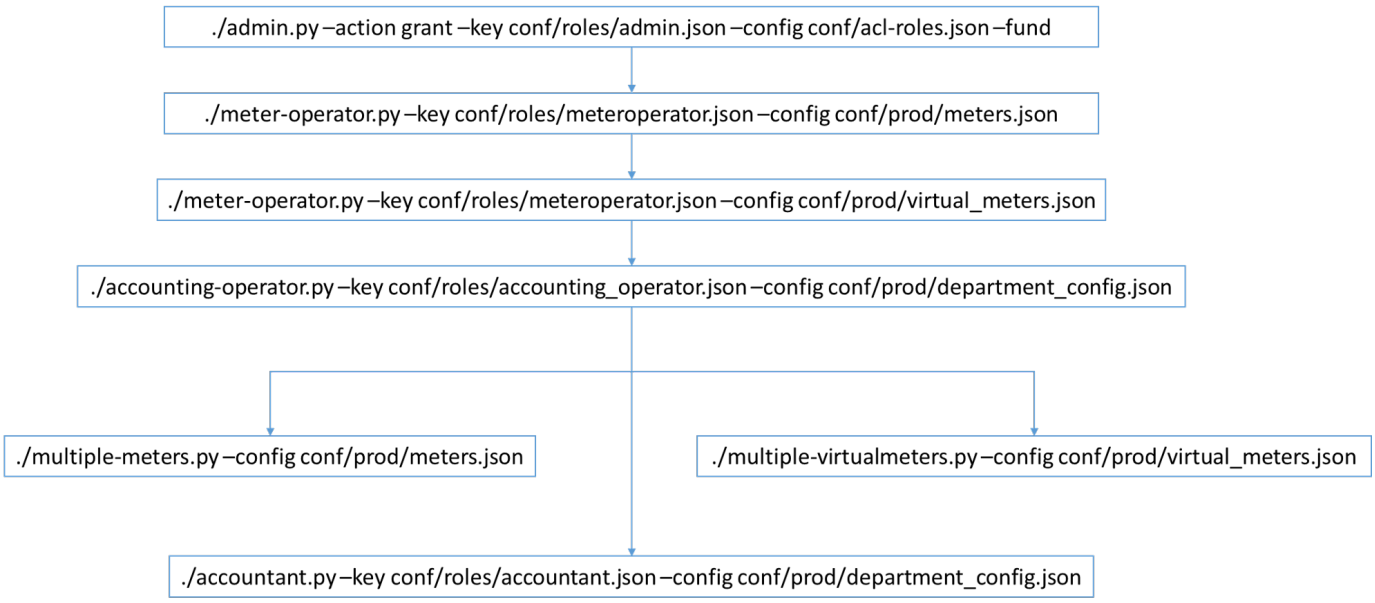


Fig. 22: The full execution process of metering and billing

XXIII. 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 running an Ubuntu 16.04 LTS Virtual Machine. The Virtual Machine was given 4 GB of RAM and used 2 Intel i5-5300U CPUs, at 2.3Ghz with 2 cores each. Due to having to launch 94 Python processes the required RAM for the simulation was consistently in the range of 2.5 GB. When the meters were not pinging, the CPU usage did not exceed 5%. When the meters are interacting with the monitoring server a 15% spike in CPU usage momentarily occurs and when the meters ping their values to the blockchain CPU usage reaches approximately 50%. This is expected due to the large amount of processes, and it is safe to assume that a device such a Raspberry Pwould not have any issue running up to 25 meter clients considering the CPU and RAM consumption required by each client. The deployment of the smart contracts was firstly tested on a local ganache[61] 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, I analyze the expected 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, I poll the monitoring server and update the values of the smart meters every 2 hours[62]. As a result, the operating cost for pinging the values of 48 meters for a year[63] is: 48 meters * 31422 gas * 0.5 readings / hmy * 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 0.016 ether per billing round for 20 departments with varying amounts of subdepartments, virtual meters and delegates, which means 0.016 ether per month and 0.192 ether per year.

XXIV. Conclusion and Future Work

XXV. Conclusion

In this Diploma Thesis I explained the functionality and applications of blockchain technology. Executing complex

[61] Software which allows setting up a local private blockchain with funded accounts and instant block confirmation times, shown in Figure 23

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

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

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. I claim that solving scalability in Ethereum requires adjustments both at the network level and at the smart contract level. I 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. I 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. I evaluated the usage of the tool Slither and found it to be superior to the other available open source auditing tools.

I 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[64] 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 XI. An access control list was developed in Chapter XIV 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 technique for storing data from Chapter VII.

XXVI. 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 my 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 my 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.

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

Appendix A
Smart Contract Development

In this Appendix I go over the steps a developer needs to go through in order to be able to code, test, debug and deploy DApps in Ethereum.

Appendix B
Setting up a development environment

In this section I explain the installation steps for the software needed for DApp development. Basic Linux knowledge is assumed. The described steps are tested in Ubuntu 17.04.

Firstly, an Ethereum client implementation must be installed on the system. I choose geth[65]. I can download the latest version of geth from https://github.com/ethereum/go-ethereum/releases.

The tools that I will be using require NodeJS and Git. I install NodeJS via the Node Version Manager[66] in order to have better management over the NodeJS versions.

I then install the tools and libraries used for having a proper development workflow. These tools are the Truffle Development Framework, Solium which is a linter to identify and fix style issues in Solidity, Web3.js which is the javascript library for interacting with the Ethereum ecosystem

Finally, I install Ganache which provides us with the ability to launch local testnet instances with prefunded accounts for local testing. Ganache has both a Graphical User Interface and a Command Line Interface version.

[65]https://github.com/ethereum/go-ethereum
[66]https://github.com/creationix/nvm

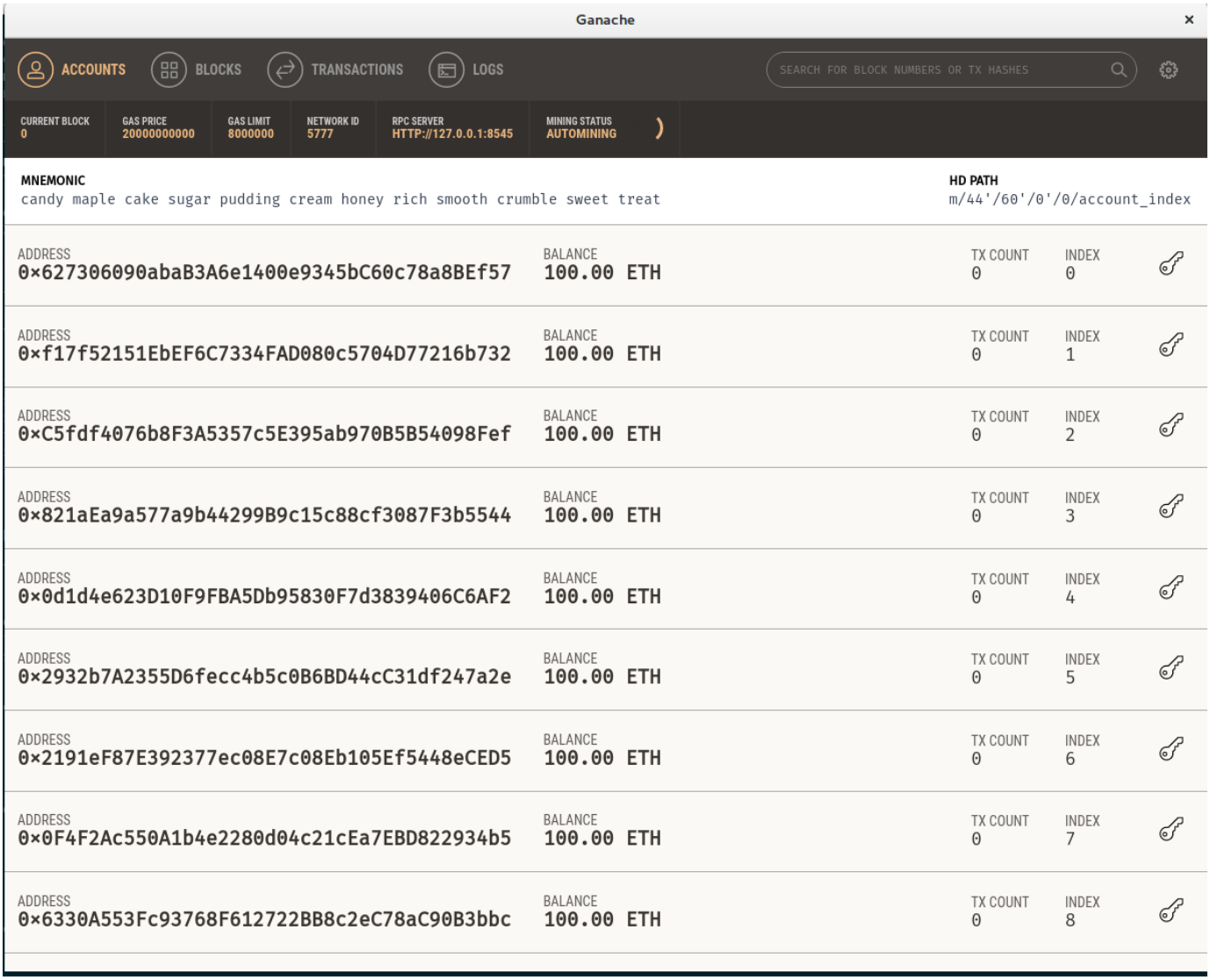


Fig. 23: Ganache User Interface

Appendix C
Interacting with Ethereum

By default, ganache starts listening at `localhost:8545` for incoming Ethereum JSON-RPC formatted messages[67], and has 10 pre-funded accounts with 100 ether, as shown in Figure 23. In addition, any transaction that gets submitted gets instantly confirmed which allows for rapid debugging. I can attach to that and instance and interact with the testnet by using the command `geth attach http://localhost:8545`.

After being connected to a new testnet instance I can query the node for the balances of each account as well as make transactions as shown in Figure 24.

After a transaction gets submitted, its transaction hash is returned, and when it is mined in a block, its transaction receipt can be retrieved which contains additional information about that transaction as shown in 25. Similarly, thet contents of a block can get fetched by querying `eth.getBlock(<number>)`.

Appendix D
Development Workflow

Truffle Framework and Ganache are the main tools used for smart contract development. When starting a new project, running the `truffle init` command results in the directory structure from Figure 26. Developed smart contracts are placed under the directory `contracts`, the deployment scripts under `migrations` and finally Mocha[68] tests under `test`.

Appendix E
Tool Development

Having understood how `geth` is used for interacting with Ethereum and Truffle for developing, deploying and testing

[67] https://github.com/ethereum/wiki/wiki/JSON-RPC

[68] https://mochajs.org/

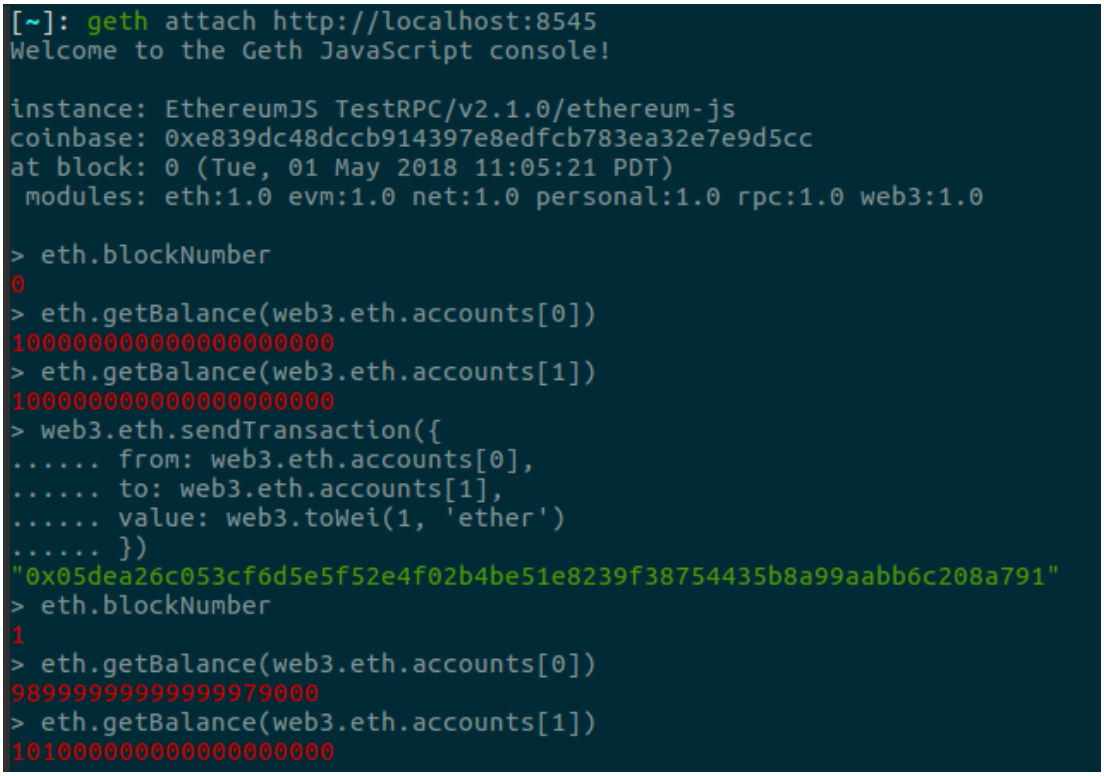


Fig. 24: Querying a node and sending a transaction

smart contracts, eventually I need to create my own tools. Web3 is the official library for interacting with Ethereum. I have already installed it for javascript, however I also use the python version for scripting client side and because I were more comfortable with tools written in Python. Web3.py is installed through pip with `pip install web3`.

In case I want to interact with a public Ethereum network, I can either host my own local node and connect to `localhost:8545` like with ganache, or I can use a third-party public full node. Currently, the most reputable free provider for this service Infura[69]. Using Infura nodes requires first obtaining an APkey. After that, interacting with a node is a matter if initializing the web3 instance, as shown in Figure 27 for Javascript and in Figure 28 for Python.

Appendix F
Scalability

Appendix G
Advantages of the Optimizer

The Solidity compiler optimizer operates on assembly. It is able to analyze the compiled code and break it down in blocks, where each instruction gets analyzed [99]. It tries to simplify expressions which can be evaluated at runtime, and 'pack' items that are less than 32 bytes long into a single storage slot, according to the following rules:

1) The first item in a storage slot is stored lower-order aligned.
2) Elementary types use only that many bytes that are necessary to store them.
3) If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
4) Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

Variable packing is very efficient because writing to storage which has been already written to before by SSTORE requires only 5000 gas instead of 20000 (I). The execution cost for `set()` in Figure 29 is 60000[70] because it initializes 3 storage slots, where it costs 45000 because it initializes 2 storage slots[71]. By using the optimizer, the third `SSTORE` is removed because the compiler can set both values with 1 `SSTORE` command in the same storage slot. As a result, the gas cost for calling `set()` with the optimizer is 40000 and requires only 2 `SSTORE` calls,where the optimizer is unable to provide any additional savings.

Appendix H
Using Libraries in Smart Contracts

Libraries are contracts which cannot hold ether and do not have storage. Using libraries results in gas savings and code reuse. According to the Solidity Documentation, 'Libraries can be seen as implicit base contracts of the contracts that use

[69]https://infura.io/

[70]The gas costs were obtained by calling the function in Remix IDE, http://remix.ethereum.org/

[71]The `uint128` items get packed in one storage slot.

```
> eth.getTransaction("0x05dea26c053cf6d5e5f52e4f02b4be51e8239f38754435b8a99aabb6c208a791")
{
  blockHash: "0xb8b5803cecac7fe2040e9dc9473136d97ee6ba7fecaf94b61a8ebebd22db9be6",
  blockNumber: 1,
  from: "0xe839dc48dccb914397e8edfcb783ea32e7e9d5cc",
  gas: 90000,
  gasPrice: 1,
  hash: "0x05dea26c053cf6d5e5f52e4f02b4be51e8239f38754435b8a99aabb6c208a791",
  input: "0x0",
  nonce: 0,
  to: "0x3b2e649925e7e1c8e980a002838b6ff6ca7e40c6",
  transactionIndex: 0,
  value: 1000000000000000000
}
>
> eth.getTransactionReceipt("0x05dea26c053cf6d5e5f52e4f02b4be51e8239f38754435b8a99aabb6c208a791")
{
  blockHash: "0xb8b5803cecac7fe2040e9dc9473136d97ee6ba7fecaf94b61a8ebebd22db9be6",
  blockNumber: 1,
  contractAddress: null,
  cumulativeGasUsed: 21000,
  gasUsed: 21000,
  logs: [],
  logsBloom: "0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000",
  status: "0x01",
  transactionHash: "0x05dea26c053cf6d5e5f52e4f02b4be51e8239f38754435b8a99aabb6c208a791",
  transactionIndex: 0
}
```

Fig. 25: Contents of an Ethereum transaction when querying a node

```
[~/Diploma/truffle-demo]: truffle init
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
[~/Diploma/truffle-demo, 2s]: tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js

3 directories, 4 files
```

Fig. 26: Directory structure after initializing a Truffle project

```
> 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
```

Fig. 27: Interacting with a node in Javascript

```
$ 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
```

Fig. 28: Interacting with a node in Python

them'. The functions of deployed Librares are called by using the opcode `DELEGATECALL`, which allows a function to be executed in the context of the caller. This results in contracts being able to delegate complex computation on an already deployed library's functionality, without having to deploy the library code. Libraries can be used to modify their associated contract's variables. Utilizing the `using` syntax, a function in a library can be called as a method of its first parameter as shown in Figure 30.

my implementation of the library for saving gas requires firstly defining the layout of variables in storage, as shown in Table XII for a Meter.

TABLE XII: Layout of variables required to store a meter in a 256 bit word

Type	Name	Index	Bits
bool	active	0	1
bytes8	id	1	64
uint48	power	65	48
uint32	timestamp	113	32
uint48	lastPower	145	48
uint32	lastTimestamp	193	32

After that, I define a `mask` variable for each size of the datatypes I use and a variable that indicates how many shifts

```
pragma solidity ^0.4.21;

contract CannotPack {
    uint128 a = 1;
    uint256 b = 2;
    uint128 c = 3;
}
```

(a) Variables cannot be packed

```
pragma solidity ^0.4.21;

contract CanPack {
    uint128 a = 1;
    uint128 b = 2;
    uint256 c = 3;
}
```

(b) Variables can be packed

```
$ solc --asm CannotPack.sol | grep sstore | wc -l
3
$ solc --optimize --asm CannotPack.sol | grep sstore | wc -l
3

$ solc --asm CanPack.sol | grep sstore | wc -l
3
$ solc --optimize --asm CanPack.sol | grep sstore | wc -l
2
```

(c) 2 SSTOREs required for the CanPack contract compared to 3 for CannotPack

Fig. 29: The optimizer saves a call to `SSTORE` when it operates on packed storage variables.

```
library CounterLib {
    struct Counter { uint i; }

    function incremented(Counter storage self) returns (uint) {
        return ++self.i;
    }
}

contract CounterContract {
    using CounterLib for CounterLib.Counter;

    CounterLib.Counter counter;

    function increment() returns (uint) {
        return counter.incremented();
    }
}
```

Fig. 30: Example of using the `using X for Y` syntax to create a `Counter` datatype [77]

must be done to set or retrieve a variable,. For example, setting `lastPower` requires shifting left 145 bits and applying a 48 bit mask,. The exposed APfor setting the value is abstracted from the end user

Appendix I
Comparing Gas Costs across Implementations

In Section X-C I made gas comparisons between the 3 developed methods, in order to determine which was the most efficient in terms of gas. The process followed for each implementation was:

1) Compile the contract with N optimizer runs, where N in $[0, 1, 100, 500, 500000]$.
2) Deploy the compiled ABand Bytecode to a local testnet[72].
3) Get gas costs for deployment.
4) Execute each of the contract functions[73].
5) Get gas costs for each function call.

[72]I used ganache as a testnet.

[73]The functions were called once since the gas costs are deterministically the same for all calls

```
    uint private constant mask48        = (1 << 48) - 1;
    uint private constant _lastPower = 1 << 145;
```

(a) Mask and shift offsets for lastPower

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

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

(b) Getting and setting a property

```
function setLastPower(bytes32 MeterData, uint48 power) internal
    pure
    returns (bytes32)
{ return
    setProperty(MeterData, mask48,  _lastPower, power);
}
function getLastPower(bytes32 MeterData) internal
    pure
    returns (uint48)
{
    return uint48(getProperty(MeterData, mask48, _lastPower));
}
```

Fig. 31: APfor setting and retrieving values by using a library

For that, I implemented a set of Python scripts responsible for automating the process. Specifically, I created:

1) A `getWeb3` script which allows connecting to an Ethereum node by providing an endpoint with the syntax `w3 = getWeb3(<endpoint>)`.
2) A `framework` script which exports a `Contract` class that can compile and deploy a smart contract when given its source code. It also provides a `call` method which allows calling a smart contract function and retrieve its gas usage.
3) A comparison script which utilizes the `framework` and the `pandas` library to create a dataframe with the gas costs for each function of the supplied smart contracts, for the specified optimizer runs.

Appendix J
Security

Appendix K
Deleting a struct with mapping

Slither provides a module for detecting a vulnerability which involves deleting a `struct` which contains a `mapping`. I provide an example of a smart contract with this flaw in Figure 32. The steps to reproduce the vulnerability are the following:

1) Create a balance for a user by calling `createBalance(idx)`, let `idx` be `42`.
2) Set the balance of a target address to a value, let the address be `0x1` and the value `100`.
3) Delete the balance by calling `deleteBalance(42)`. It is expected that the `struct` at the index 42 will be deleted, along with all its contents.
4) Create a balance for another user by calling `createBalance(42)`. Retrieving that user's balance it can be seen that it is 100, instead of the expected 0.

```
contract Bank {

    struct BalancesStruct{
        address owner;
        mapping(address => uint) balances;
    }

    mapping(uint => BalancesStruct) stackBalance;
    function createBalance(uint idx){
        require(stackBalance[idx].owner == 0);
        stackBalance[idx] = BalancesStruct(msg.sender);
    }

    function deleteBalance(uint idx){
        require(stackBalance[idx].owner == msg.sender);
        delete stackBalance[idx];
    }

    function setBalance(uint idx, address addr, uint val){
        require(stackBalance[idx].owner == msg.sender);

        stackBalance[idx].balances[addr] = val;
    }

    function getBalance(uint idx, address addr) returns(uint){
        return stackBalance[idx].balances[addr];
    }

}
```

Fig. 32: Mapping inside a struct

APPENDIX L
HONEYPOT SMART CONTRACTS

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

The contract in Figure 33 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[74], 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.

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[75] which looks for the value of `owner` in the `Owned` contract. As a result, the 'real' `owner` of the contract is unchanged, and the attacker is unable to claim ownership, effectively losing their funds. Running

[74]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.

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

```
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;
    }
}
```

Fig. 33: A honeypot which takes advantage of Variable Shadowing.

TxHash	Block	Age	From		To	Value
0x11713ad67126f7...	4755628	116 days 19 hrs ago	0x00a0d727fa2a08e...	IN	0x4dc76cfc65b14b3...	0.42 Ether
0xbbe67e38e60611...	4755591	116 days 19 hrs ago	0x00a0d727fa2a08e...	IN	0x4dc76cfc65b14b3...	0.42 Ether
0x577e1726283c2e...	4749448	117 days 20 hrs ago	0x22dc0137012998...	IN	0x4dc76cfc65b14b3...	0.21 Ether
0xfd59f1b4c34e232...	4527013	154 days 20 hrs ago	0x125d657d5cd16bf...	IN	0x4dc76cfc65b14b3...	0.100000000001 Ether
0x530a8ba6a4e7f07...	4525157	155 days 4 hrs ago	0x80028f80c7d5959...	IN	0x4dc76cfc65b14b3...	0.1 Ether
0x50f803e00f5e782...	4525139	155 days 4 hrs ago	0x80028f80c7d5959...	IN	Contract Creation	0 Ether

Fig. 34: Transactions trying to get ownership of the honeypot contract

Slither on the contract can reveal the variable shadowing as shown in 35.

APPENDIX M
IMPLEMENTATION

APPENDIX N
GAS COSTS COMPARISON

In order to compare the gas consumption between implementations during testing I use eth-gas-reporter[76] which allows for to get gas usage per unit test, as well as retrieve average gas usage per method called. I configure `eth-gas-reporter` in the `truffle-config.js` file, and by prefixing a `truffle test` command with `GAS_REPORTER=1` I can get enhanced tests as shown in Figure 36. The metrics shown in Table XI are obtained after running the 4 tests in the supplied `smartcontract-gascomparison` repository.

APPENDIX O
ACCESS CONTROL

Modifiers in Solidity are used to change the behavior of functions, similarly to decorators in Python[77]. The modifier in the case of Figure 3738 gets called before the call to `foo()`, and after the `require` statement gets verified, execution resumes from '_'.

[76]https://github.com/cgewecke/eth-gas-reporter

[77]https://www.python.org/dev/peps/pep-0318/

Fig. 35: Slither is able to detect the variable shadowing in `owner`

```
$ GAS_REPORTER=1 truffle test test/testMeterStructs.js

Using network 'test'.


  Contract: MeterDB.sol
    ✓ Activates a meter and pings a value (1043241 gas)

·-------------------------------------------------------------|-----------------------------·
|                             Gas                             ·  Block limit: 6721975 gas   |
·-----------------------------------------|-------------------|-----------------------------·
|  Methods                                ·        3 gwei/gas ·       681.61 usd/eth       |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  Contract       ·  Method               ·  Min  ·  Max  · Avg ·  # calls    ·  usd (avg)    |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  MeterDBStructs ·  activateMeter        ·    -  ·    -  · 134274 ·          1  ·        0.27   |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  MeterDBStructs ·  activateMeterBatch   ·    -  ·    -  ·     - ·          0  ·           -   |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  MeterDBStructs ·  deactivateMeter      ·    -  ·    -  ·     - ·          0  ·           -   |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  MeterDBStructs ·  ping                 ·    -  ·    -  · 57844 ·          2  ·        0.12   |
·-----------------|-----------------------|-------|-------|---|-------------|---------------·
|  Deployments                            ·                   ·  % of limit ·                |
·-----------------------------------------|-------|-------|---|-------------|---------------·
|  MeterDBStructs                         ·    -  ·    -  · 793279 ·     11.8 % ·        1.62   |
·-----------------------------------------|-------|-------|---|-------------|---------------·

  1 passing (8s)
```

Fig. 36: Running a truffle test with eth-gas-reporter

```
contract Owned {
    address owner;
    function Owned() public { owner = msg.sender; }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}

contract Test is Owned {
    function foo() onlyOwner {
        /* do something only owner can do */
    }
}
```

Fig. 37: Using the onlyOwner modifier to enforce access control for a function

The usage of modifiers has been particularly widespread on enforcing access control to create contracts where only specific addresses can access selected privileged functions of a contract.

In the naive case, if there were a number of smart contracts deployed by the same entity which had privileged functions callable only by an owner, the same pattern would have to be used in each contract. If a developer wanted to make the ownership check more flexible, they would have to modify the privileged members in each contract. This does not scale well, due to the number of transactions involved. In addition, if there were multiple roles, managing them on different contracts can become inefficient and has increased room for error.

I use the implementation of an Access Control List contract from DAppHub[78]. This pattern involves deploying two contracts, where one gets inherited by any contract that wishes to have access control, and the other stores all the access control list logic.

`DSRoles.sol` allows the creation of up to 256 roles, where each role can be granted permission to call the function of a contract.

```
function canCall(address caller, address code, bytes4 sig)
    public
    view
    returns (bool)
{
    if( isUserRoot(caller) || isCapabilityPublic(code, sig) ) {
        return true;
    } else {
        bytes32 has_roles = getUserRoles(caller);
        bytes32 needs_one_of = getCapabilityRoles(code, sig);
        return bytes32(0) != has_roles & needs_one_of;
    }
}
```

Fig. 38: `canCall` in `DSRoles.sol` checks if the sender is authorized to perform the function call according to the permissions set by the ACL

In order for a contract to use the ACL (`authority`),

[78]https://github.com/dapphub/ds-roles, https://github.com/dapphub/ds-auth

it needs to be linked with it by the owner by inheriting `DSAuth.sol` and calling the `setAuthority` function. Afterwards, any function that gets modified with `auth` will ask for permission from the linked instance of `DSRoles.sol` as shown in Figure 39. This allows for a modular approach where an administrator[79] can grant and revoke permissions from users, depending on their clearance level.

```
modifier auth {
    require(isAuthorized(msg.sender, msg.sig));
    _;
}

function isAuthorized(address src, bytes4 sig) public view returns
    (bool) {
    if (src == address(this)) {
        return true;
    } else if (src == owner) {
        return true;
    } else if (authority == DSAuthority(0)) {
        return false;
    } else {
        return authority.canCall(src, this, sig);
    }
}
```

Fig. 39: Using the auth modifier to request permission from the linked authority (ACL)

Appendix P
Listening for Events

Events are an inexpensive way to store data in the Ethereum blockchain. An `Event` is a special kind of datatype that gets logged in the transaction receipt after a transaction gets mined. As a result, if I call a function that emits an event, that event is stored permanently in the blockchain for approximately 2000 gas. Listening for events is a fundamental process that web applications do in order to interact with a smart contract. I can also fetch all past events that a smart contract has emitted, which allows us to have a full historical log over some action that has happened in a smart contract.

For my implementation I implemented a proof of concept script written in Javascript which can fetch all the readings that have been pinged in the smart contract, and additionally filter them by Meter Id. The same script can be implemented in Python, however there are ongoing issues with it when interacting with ganache in web3.py[80]. An example is shown in Figure 40 and the implementation is in Figure 41.

```
$ ./get_meter_readings --rpc http://localhost:8545 --address "0
    xb549d25374337a3d2cc7bad7a4009d8cb1c3f537" --meter "ELT01 ELT02"
Meter ELT01: 10 kWh at 1525281846
Meter ELT01: 20 kWh at 1525285175
Meter ELT02: 15 kWh at 1525281852
Meter ELT02: 22 kWh at 1525285166
```

Fig. 40: Retrieving all the historical readings for meters ELT01 and ELT02

```
#!/usr/bin/env node
const Web3 = require('web3')
const args = require('optimist').string("address").argv;
const truffleContract = require("truffle-contract");
const web3 = new Web3(new Web3.providers.HttpProvider(args.rpc));

function toAscii(input) {
    return web3.toAscii(input).replace(/\u0000/g, '');
}

let conf;
if (args.abi) {
    conf = require(args.abi);
} else {
    conf = require('./conf/abi/MeterDB.json');
}

let address = args.address;
let abi = conf.abi;

const contract = truffleContract({ abi });
contract.setProvider(web3.currentProvider);
meterdb = contract.at(address);

let meters = {}
if (args.meter) {
    meters = { "meterId": args.meter.split(" ") };
}

meterdb.Pinged(meters, { fromBlock: 0, toBlock: 'latest' }).get((
    err, res) => {
    if (!err) {
        res.forEach(log => {
            // event Pinged(address indexed meterAddress, bytes8
               indexed meterId, uint80 power, uint80 timestamp);
            arg = log.args;
            meterId = arg.meterId;
            power = arg.power;
            timestamp = arg.timestamp;
            console.log('Meter ' + toAscii(meterId) + ': ' + power
               + ' kWh at ' + timestamp);
        });
    }
});
```

Fig. 41: Script to retrieve all meter readings

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] "Colored coins," http://coloredcoins.org/, 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, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342
[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. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1107.html#abs-1107-4524
[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. [Online]. Available: http://arxiv.org/abs/1708.04748
[17] "Eth price stats and information," https://bitinfocharts.com/ethereum/.
[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/.

[79]The version I use contains a centralized entity which handles permission. This can be re-implemented depending on the desired governance model

[80]https://github.com/trufflesuite/ganache-cli/issues/494

[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*. Springer, 1992, pp. 139–147.

[25] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," 12 2013. [Online]. Available: https://eprint.iacr.org/2013/881.pdf

[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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 3–16. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978341

[27] P. Kasireddy, "How does ethereum work, anyway?" https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369, 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] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," *Unknown*, 2018.

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

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

[38] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," http://lightning.network/lightning-network-paper.pdf, 2016.

[39] "Raiden network," https://raiden.network/.

[40] "Funfair technologies," https://funfair.io.

[41] "Raiden network," https://counterfactual.com/.

[42] 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.

[43] "Loom network," https://loomx.io.

[44] "Cosmos network," https://cosmos.network.

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

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

[47] M. Vukolić, "Rethinking permissioned blockchains," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, ser. BCC '17. New York, NY, USA: ACM, 2017, pp. 3–7. [Online]. Available: http://doi.acm.org/10.1145/3055518.3055526

[48] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," *CoRR*, vol. abs/1703.03994, 2017. [Online]. Available: http://arxiv.org/abs/1703.03994

[49] 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: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

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

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

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

[53] 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

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

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

[56] "Smartcheck," https://tool.smartdec.net/.

[57] "Securify," https://securify.ch/.

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

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

[60] https://twitter.com/dguido/status/966795184603312130.

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

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

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

[64] "Grid+," https://gridplus.io/.

[65] "Powerledger," https://powerledger.io/.

[66] "Brooklyn microgrid," https://brooklyn.energy.

[67] 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. [Online]. Available: https://doi.org/10.1007/s00450-017-0360-9

[68] "Mining," https://github.com/ethereum/wiki/wiki/Mining.

[69] "Sharding faq," https://github.com/ethereum/wiki/wiki/Sharding-FAQ.

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

[71] 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.

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

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

[74] "Solc.acomedyinoneact," https://github.com/figs999/Ethereum/blob/master/Solc.aComedyInOneAct.

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

[76] "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/.

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

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

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

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

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

[82] 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.

[83] 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.

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

[85] "Ethereum classic," https://ethereumclassic.github.io/.

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

[87] "Parity multisig wallet exploit hits swarm city funds - statement by the swarm city core team."

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

[89] "Eip-999: Restore contract code at 0x863df6bfa4," https://ethereum-magicians.org/t/eip-999-restore-contract-code-at-0x863df6bfa4/130.
[90] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *CoRR*, vol. abs/1710.06372, 2017. [Online]. Available: http://arxiv.org/abs/1710.06372
[91] "Use our suite of ethereum security tools," https://blog.trailofbits.com/2018/03/23/use-our-suite-of-ethereum-security-tools/.
[92] "New opcode: Staticcall," https://github.com/ethereum/EIPs/pull/214, 2017.
[93] "Ethernaut," https://ethernaut.zeppelin.solutions/.
[94] "Hack this contract," http://hackthiscontract.io/.
[95] "Capture the ether," https://capturetheether.com/.
[96] E. Comission, "2030 energy strategy," https://ec.europa.eu/energy/en/topics/energy-strategy-and-energy-union/2030-energy-strategy.
[97] EY, "Blockchain in power and utilities: real or hype?" 2017.
[98] "Technical introduction to events and logs in ethereum," https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e.
[99] "Solidity documentation," http://solidity.readthedocs.io/en/v0.4.21/miscellaneous.html.