

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

Aristotle University of Thessaloniki

Honda R&D Europe

Georgios Konstantopoulos

Contents

1	Abstract	4
2	Introduction	5
2.1	History	5
2.2	Problem Statement	5
2.3	Scope	5
2.4	Outline	6
3	Ethereum and Blockchain Basics	7
3.1	Overview	7
3.1.1	Public Key Cryptography	7
3.1.2	Cryptographic Hash Functions	8
3.2	Ethereum Blockchain	8
3.2.1	Transaction	8
3.2.2	Block	9
3.2.3	Blockchain	10
3.2.4	Blockchain Types	10
3.2.5	Transactions as State Transitions	11
3.2.6	Ethereum Virtual Machine	12
3.2.7	Gas	12
3.3	Programming in Ethereum	13
3.3.1	Programming Languages	13
3.3.2	Tooling	14
4	Blockchain Scalability	16
4.1	Bottlenecks in Scalability	16
4.2	Network Level Scalability	16
4.3	Contract Level Scalability	17
4.3.1	Gas Costs	17
4.3.2	Gas Savings Case Study	19
4.3.3	Results	21
5	Smart Contract Security	22
5.1	Past Vulnerabilities	22
5.2	Security of deployed Smart Contracts	22
5.3	Best Practices	22
5.4	Access Control	22

6	Blockchain and the Energy Market	23
6.1	Advantages of Blockchain	23
6.2	Our Use-case	23
7	Design and Implementation	24
7.1	Business Logic	24
7.2	Smart Contracts	24
7.3	Monitoring Server	24
7.3.1	REST API	24
7.3.2	Python Client	24
7.3.3	web3.py interaction	24
8	Conclusion	25
8.1	Results	25
8.2	Future Work	25

1 Abstract

We leverage the power of Smart Contracts to create a pilot energy use-case on Ethereum. We propose a suite of Smart Contracts which can be utilized to trustlessly store and verify the kilowatthour readings of an arbitrary number of ‘smart-meters’ on the Ethereum network, while also applying accounting computations in order to properly bill that energy to the corresponding departments of a company structure. Finally, contributions towards smart contract security and scalability are made.

2 Introduction

2.1 History

In 2009 Satoshi Nakamoto published the Bitcoin whitepaper [17]. There, Nakamoto describes ‘a purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution.’ In the beginning, Bitcoin was primarily used for fast and low-cost financial transactions. It was soon realized that its uses could be extended to more than just transferring value from A to B. The concept of colored coins, [3] was introduced, where users were able to embed extra data on a bitcoin which resulted in coins that could represent ownership over a land title or a domain name. In 2015, Vitalik Buterin authored the Ethereum Whitepaper [7] which was an alternative cryptocurrency to Bitcoin that enabled the creation of *smart Contracts*. smart Contracts as a term was first introduced by Nick Szabo in 1996 [19] as a model for verified trustless computation. The Ethereum Network acts as a world computer and smart contracts are code that gets executed trustlessly on every node that is part of the network.

2.2 Problem Statement

The problem this Master Thesis solves is: How an entity can manage the energy consumed by a complex system of energy meters. The system should be able to bill and perform accounting on the metering data, based on a pre-specified accounting model which can be changed at runtime. The system must be transparent, distributed, decentralized, easy-to-use and secure. Anyone in the network should be able to verify the validity transactions. It also needs to be scalable at reasonable cost.

2.3 Scope

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

2.4 Outline

Chapter X describes Y

3 Ethereum and Blockchain Basics

3.1 Overview

Before getting into the specifics of blockchains and Ethereum, the next section will be used to explain fundamental terms on cryptography and blockchain.

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

Any future reference to blockchain terminology such as the contents of a block or a transaction will be referring to the implementations of the Ethereum Platform. The Ethereum Yellowpaper provides details on the formal definitions and contents of each entity [21].

3.1.1 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 usually called **public** and **private**, due to the private key being known only to its owner while the public key is known to the public. The main advantage of Public Key Cryptography is the lack of need for a secure channel for the initial exchange of keys between any communicating parties.

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

When a person encrypts a message with one key, its pair can be used to decrypt the same message. If a message gets encrypted with the private key of the sender, any receiver can verify that the message was indeed sent by the sender as they are the only possible owners of the private key used to encrypt the message. This achieves authentication, and the process is often referred to as *signing* of a message.

3.1.2 Cryptographic Hash Functions

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

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

1. **Collision Resistance:** It should be computationally infeasible to find x and y such that $H(x) = H(y)$.
2. **Pre-Image Resistance:** Given $H(x)$ it should be computationally infeasible to find x .
3. **Second Pre-Image Resistance:** Given $H(x)$ it should be computationally infeasible to find x' so that $H(x') = H(x)$.

Bitcoin uses the SHA-256 cryptographic hash function, while Ethereum uses KECCAK-256. Both functions' outputs are 256 bits long which is considered secure given the document's writing date standards.

It should be noted that although similar, a second preimage attack is significantly more difficult than a preimage attack due to the attacker being able to manipulate only one input in order to cause a collision.

3.2 Ethereum Blockchain

3.2.1 Transaction

The contents of a transaction in Ethereum are as follows:

[illegible]


```

16     0
17   ]
18 },
19   v: '0x1c',
20   r: '0
      xe7df9fee346ea3575dd541e120df7b154d46bba2bf859352b29a7f2c6dd5fab3
      ',
21   s: '0
      x68d266b140ac468ee328592ca6b4a461edace859d6837358bc3f8eeeb24ade63
      ',
22 }

```

The fields `blockHash`, `blockNumber`, `from`, `to`, `value`, `hash` are all usual fields that can be found in blockchain transactions. `Gas` and `gasPrice` are used for performing computations on the Ethereum platform and will be analyzed in Section X. Ethereum uses an account model compared to Bitcoin's UTXO model [CITE]. In the account model, an attacker can replay a transaction by rebroadcasting it. This is mitigated by adding a nonce to each account's transactions which gets incremented after each transaction. The input field allows embedding extra data to a transaction. This can be used either to add a message or in the case of smart contracts, to have the contents of a function call. The `v`, `r`, `s` parameters are outputs from the Elliptic Curve Digital Signature Algorithms

3.2.2 Block

A block is a data structure which is comprised of data forming the block header and transactions. The interesting field for the thesis is the `gasLimit` field which will be discussed like the transaction `gasLimit` in section X.

```

1 > web3.eth.getBlock(5284738)
2 { difficulty: BigNumber { s: 1, e: 15, c: [
3     32,
4     85319757566868
5   ]
6 },
7   extraData: '0x7869786978697869',
8   gasLimit: 7995219,
9   gasUsed: 1547361,
10  hash: '0
      x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
      ',
11  logsBloom: '0
      x000208000002040002000400000000000010040000000000080002000000008400080040022
      ',
12  miner: '0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb',
13  mixHash: '0
      x29b6efa55ad0298b0c90f21e9e23d572977ffb3c5064a9816a69bb2bf2a9effd
      ',
14  nonce: '0xabed128000fed25e',
15  number: 5284738,
16  parentHash: '0
      xb7063b9c7b05c95c35a329717e44875829cc740b2e0749e03d54806dcf34b520
      ',
17  receiptsRoot: '0
      xe5e176557b9f40394917191095b706a2a331742f0dc93a10e1d59b5e297ee0b5
      ',

```

```
18   sha3Uncles: '0
      x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
      ',
19   size: 7789,
20   stateRoot: '0
      x1c62917ac72a2b76e00053efbb7af0d6949e86cafb3f983812d763715c6c9905
      ',
21   timestamp: 1521484243,
22   totalDifficulty: BigNumber { s: 1, e: 21, c: [
23     31406307,
24     78318927526632
25   ]
26 },
27   transactions: [ '0
      x6a5d9e470bbff3eb476e20647fbe66e0cec7795291efd6301e6028865d0d4201
      ',
28     '0
      xbe1c3e767e34d5d668ea50d3400b2e11a663479f931c225eda5e1d314e012589
      ', ...
29 ],
30   transactionsRoot: '0
      xb0a066469d74fe1f450c5fa8a1f59c5b7305feb6336d0d59f347a2b2c7a8c579
      ',
31   uncles: []
32 }
```

3.2.3 Blockchain

Blocks refer to the previous blocks recursively until the genesis block (first block in existence), forming a chain of blocks. The set of rules which allow an actor to add a valid block to the blockchain is called a *consensus algorithm*. In order to have consensus in distributed systems, all participating nodes must have the same version (often called history) of the system (blockchain). A malicious node could create an arbitrary block crediting them with any amount of Ether. In order to avoid that, consensus algorithms elect a network participant to decide on the contents of the next block, in a fair manner. This process is often called mining, due to the popularity of the Proof of Work consensus algorithm which allows nodes called *miners* to propose a new block if they solve a hard to compute problem. Consensus algorithms require their problem to be easy to verify however, as when somebody solves the problem and broadcasts it to the network, its solution must be verified swiftly in order to get accepted and propagated to the rest of the network, or rejected. The process of mining and how consensus is achieved is considered outside the scope of this Master Thesis.

3.2.4 Blockchain Types

Blockchains are inspectable and public. Any entity can setup a node, download the full blockchain history and view all the transactions caused by anyone participating in that network. This is one of the main benefits of using a blockchain, transparency.

We categorize blockchains in two kinds (different authors might have different classifications):

1. Public or Permissionless: Low barrier to entry, transparent and immutable.



Figure 3.1: Blockchain = chain of blocks

2. Private or Permissioned: Federated participation, can obscure certain pieces of data, ability to modify and revert past transactions if needed.

Vitalik Buterin goes indepth in the advantages and disadvantages between private and public blockchains in [6]. Due to the scalability and privacy restrictions of public blockchains, corporations that are looking to include blockchain technology in their processes are looking for a solution NOW, when the research and development is still not at that level. As a result, permissioned blockchains as JP Morgan's Quorum [15], Hyperledger or Corda have arised, with aims to solve these problems.

3.2.5 Transactions as State Transitions

In Bitcoin, the state is created through the Unspent Transaction Outputs set, which defines the amount of BTC a user can spend. As Ethereum does not use UTXO but an account model, there needs to be a way to monitor the state. This is done through a data structure called Patricia State Trie [16] which provides an efficient mapping of key and value pairs where the key is the address of the Ethereum account and the value is the Recursive Length Prefix encoding [11] of the account's data.

Whenever a transaction gets successfully mined, the world state gets updated accordingly.

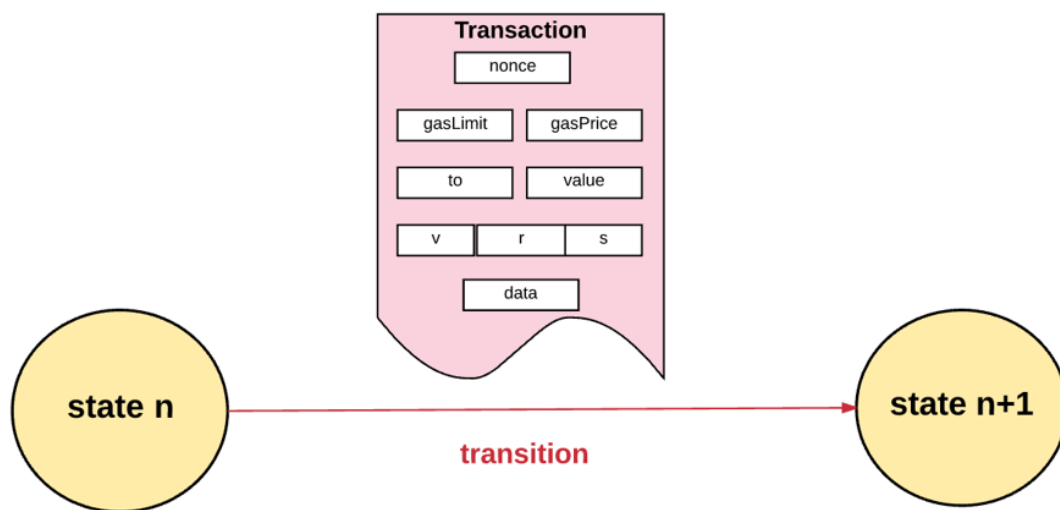


Figure 3.2: After each transaction the state gets updated

3.2.6 Ethereum Virtual Machine

The Ethereum Virtual Machine 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 (discussed in Section X). Since all nodes redundantly process all transactions and contract executions this process, this can be used by an attacker to maliciously flood the network with transactions and cause multiple computers around the world to perform costly computations forever. There needs to be a computational cost. In Ethereum it is called gas and to the end user it manifests itself as the fees needed for a transaction (be it value transfer or contract call) to complete successfully.

3.2.7 Gas

Every computational step on Ethereum costs gas. The simplest transaction which involves transferring Ether from one account to another costs 21000 gas. Calling functions of a contract involves additional operations whose costs can be estimated through the costs described in [4, 21].

[INSERT GAS / WEI / ETHER denominations]

When referring to blocks, the *gasLimit* is the cumulative gas that is needed by all transactions included in that block. This is analogous to the block size in Bitcoin and effectively limits the amount of transactions that can be confirmed per block.

Every unit of gas costs a certain amount of *gasPrice* which is set by the sender of the transaction. It is the case that:

$$totalTransactionCost = gasPrice * gasLimit \quad (3.1)$$

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

This effectively creates a fee market where actors increase the *gasPrice* value to have them confirmed faster. In the times of network congestion such as popular Initial Coin Offerings [CITE] or mass-driven games such as CryptoKitties [CITE], the network has become very expensive to use and oftentimes unusable with transactions taking hours to confirm [CITE].

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

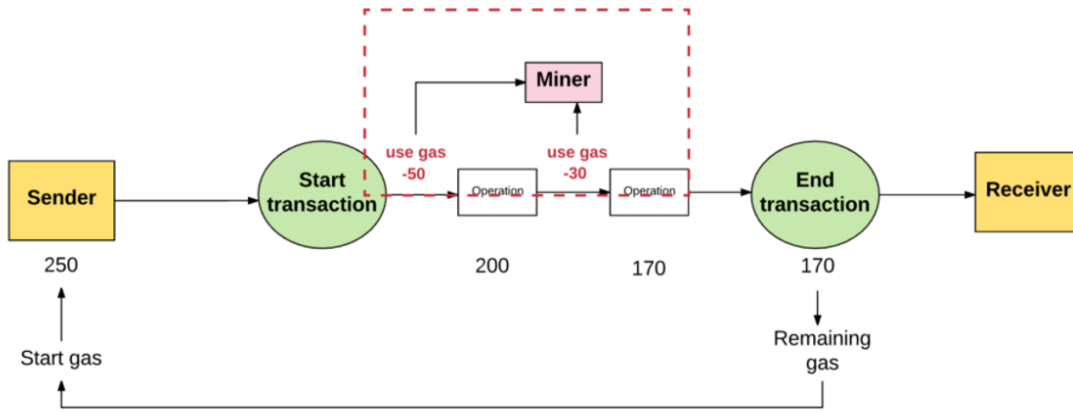


Figure 3.3: Successful transaction (from [13])

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 changes that would happen are reverted. This is similar to the SQL transaction commit-rollback pattern.

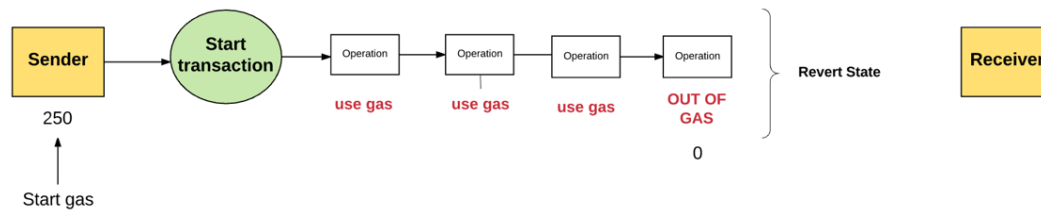


Figure 3.4: Out of gas transaction (from [13])

3.3 Programming in Ethereum

The EVM has its own language called EVM bytecode [REPHRASE]. Programmers often write in higher-level languages such as Solidity and then compile to lower level code.

3.3.1 Programming Languages

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

Solidity is the most supported language in the ecosystem and although often comparable to Javascript, we argue that Smart Contracts remind more of C++ or Java, due to their object oriented design.

Due to the nascence of these languages and the security mistakes that have occurred due to them providing programmers with powerful state-changing functions,

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

Figure 3.5: Basic Solidity Smart Contract

there is also progress towards creating functional programming languages such as Bamboo or ones that are formally verifiable.

A compiler is needed to output both the EVM Bytecode and the Application Binary Interface (ABI) so that a third party library can interact with the Smart Contract.

3.3.2 Tooling

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

Client (Node) Implementations and Testnets

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

Smart contracts are immutable once deployed which means that their code cannot change. In addition, they also cost to be deployed, which means that development can get expensive and inefficient. For that, public test networks (testnets) exist which allow for testing free of charge. Kovan and Rinkeby are functioning with the Proof of Authority [20] consensus algorithm, compared to Ropsten running the Proof of Work [10] which is the same as the Ethereum main network's (with less difficulty).

Comparison between test networks:

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

In addition, before deploying to a testnet, developers are encouraged to run their own local testnets in order to further their development processes. Geth and Parity allow for setting up private testnets, however the go-to tool for this process is ganache¹ (formerly known as testrpc).

Web3

Web3 is the library used for interacting with an Ethereum node. The most feature-rich implementation is Web3.js² 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.py³. We showcase an example of connecting and fetching the latest block from Ropsten and Mainnet using Web3.js and Web3.py. The full specifications of each library's API can be found in their documentation⁴⁵

```
1 $ node
2   > Web3 = require('web3')
3   > INFURA_API = process.env.INFURA_API // Infura is a third
      party service that allows us to connect to their
      Ethereum node without setting up our own.
4   > web3 = new Web3(new Web3.providers.HttpProvider("
      https://mainnet.infura.io/" + INFURA_API));
5 > web3.eth.blockNumber
6 5289236
```

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

Truffle

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

Docker

Explain docker...

¹<http://truffleframework.com/ganache>

²<https://github.com/ethereum/web3.js>

³<https://github.com/ethereum/web3.py>

⁴<https://github.com/ethereum/wiki/wiki/JavaScript-API>

⁵<https://web3py.readthedocs.io/en/stable/>

4 Blockchain Scalability

4.1 Bottlenecks in Scalability

A blockchain's scalability is often measured in transactions per second. A block gets appended to the chain every 15 seconds on average in Ethereum, and can contain only a finite amount of transactions. As a result, transaction throughput is bound by the frequency of new blocks and by the number of transactions in them.

Proof-of-Work has been the only consensus algorithm to date that has proven to be effective against attackers, while still relatively maintaining the decentralization of the network. Due to faster block times, it can handle 7 transactions per second (tx/s), which is better than bitcoin's 3 tx/s however still not comparable to Visa's 2000 consistent tx/s or 60000 at peak. As a result, creating scalable blockchain architectures has been a topic of interest.

We argue that there are two levels of scalability, scalability on contract and on network level. Better contract design can result in transactions which require less gas to execute, and thus allow for more transactions to fit in a block while also making it cheaper for the end user. It should be noted that as Ethereum's current `blockGasLimit` is set by the miners at 8003916, if all transactions in Ethereum were financial transactions (each costing 21000 gas), each block would be able to handle 381 transactions per block, which is 25 tx/s, which is still not comparable to traditional payment operators.

4.2 Network Level Scalability

A naive solution in achieving scalability involves increasing the size of each block, or in Ethereum terms the *blockGasLimit*. This solution is not sustainable as it compromises decentralization, due to increased network and hardware costs. Bigger blocks require more disk space for storing the blockchain, better bandwidth for the block propagation and more processing power on a node to verify the computations in a block. 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. The `blockGasLimit` can be voted on by miners¹.

Long term solutions in Ethereum are categorized in:

1. Sidechains: First described in [5], sidechains (side-blockchains) are running in 'parallel' to the mainchain, while using a sort of mechanism to benefit from the security of the main blockchain (mainchain). This allows them to

¹<https://www.etherchain.org/tools/gasLimitVoting>

implement consensus rules which are more flexible and customized to their use-case, allowing for potentially infinite scalability.

2. **Proof of Stake:** Proof of Work provides security at the cost of energy consumed by miners. Alternative consensus algorithms such as Proof of Stake (PoS), are more friendly to the environment than Proof of Work[9]. Instead of consuming energy, PoS involves validators who are ‘staking’ their ETH and by modeling their incentives and appropriately punishing malicious behavior, the network can have the same security as PoW. Ethereum is planning to transition to PoS, however there is no clear date when this is going to happen as this is still under heavy research.
3. **Sharding:** Due to the architecture of the EVM, it is not parallelizable. Sharding refers to having nodes which validate different parts of the blockchain, allowing for parallelizability on computations.
4. **State channels:** In the case of micropayments which involve a number of transactions between parties, there is no need to make every transaction on the blockchain. The proposed technique involves exchanging signed messages off-chain and settling when the transactions are finished. As a result, a transaction is done to open the state channel, and another to settle it.

4.3 Contract Level Scalability

In a recent study [8], after evaluating 4240 smart contracts, it is found that over 70% of them are under-optimized with respect to gas from the compiler. In this section we explore how gas gets computed and ways we can save on gas and transaction costs.

4.3.1 Gas Costs

An Ethereum transaction’s gas costs are split in:

1. **Transaction Costs:** The cost of sending data to the blockchain. There are 4 items which make up the full transaction cost:
 - (a) The base cost of a transaction (21000 gas)
 - (b) The cost of a contract deployment (32000 gas)
 - (c) The cost for every zero byte of data or contracts for a transaction.
 - (d) The cost of every non-zero byte of data or contracts for a transaction.
2. **Execution Costs:** The cost of computational operations which are executed as a result of the the transaction, as described in detail in [21, 4]

[INSERT TABLE ON GAS COSTS SHOWING SSTORE ETC]

Gas costs get translated to transaction fees. As a result, a contract should be designed to minimize its operational gas costs in order to minimize its transaction fees. In addition, as gas is a unit for computational costs, less gas consumed results in

less burden on the nodes validating the smart contracts which can lead to increased scalability.

From the gas cost table, it can be seen that the most expensive operations involve SSTORE operations. The focus of this section will be to explore ways to decrease gas costs on Smart Contracts, either through better practices or by handcrafting optimizations for specific use cases.

It should be noted, that non-standard methods have been proposed for reducing fees incurred by gas costs. A recent construction[14] describes a method for buying gas at low cost periods and saving it in order to spend it when prices are higher at times when the network is congested. The economic applications of gas arbitrage are outside the scope of this Master Thesis.

General rules that should be followed for saving gas costs:

1. Enable compiler optimizations (although can lead to unexpected scenarios [2])
2. Reuse contracts through libraries[12]
3. Setting a variable back to zero refunds 15000 gas through SSTORE, so if a variable is going to be unused it is considered good practice to call delete on it.
4. Use 'bytes32' instead of 'string' for strings that are of known size. 'bytes32' always fit in an EVM word, while 'string' types can be arbitrarily long and thus require more gas for saving their length. [NEEDSCITATION]
5. Do not save large amounts of data on a blockchain. Instead, save a hash to prove their existence at a lower point in time.

As described in [8] there is a lot of room for further compiler optimizations. Future Solidity compiler versions are addressing some already²³⁴

The EVM operates on 32 byte words [CITE] implying that a SSTORE command is needed to store 32 bytes of data. 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.

²<https://github.com/ethereum/solidity/issues/3760>

³<https://github.com/ethereum/solidity/issues/3716>

⁴<https://github.com/ethereum/solidity/issues/3691>

```

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

```

```

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

```

Figure 4.1: Running the optimizer in storage variables less than 256 bytes results in 2 SSTORE commands instead of 6 which a significant saving in gas costs

4.3.2 Gas Savings Case Study

In order to illustrate our findings and compare accross different scenarios, we will perform a Solidity benchmarking test based on a use-case of a Solidity Smart Contract which describes a game. The contract must allows a user to register as a player if they are not already registered and allow them to create a character with some traits. Table X describes these traits:

Table 4.1: Required variables and size. Sizes add up to 256 bits

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	bytes19	Utilize the rest of the word for metadata

The size of the variables is selected so that all the information required to describe a ‘Character’ can fit in a 256 bit word.

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

1. Tightly packed structures for setting data

2. Bit masking encoding for setting data
3. Bit masking encoding utilizing libraries, influenced by [18].

The full contracts of each contract can be found in the Appendix. For each test described, the optimizer was run 0, 1, 100, 500 and 50000 times.

GameTightlyPacked.sol

We're utilizing a structure here and by taking advantage of the optimizer packing everything in a word we can perform a full write to structure with only X gas

GameByteMasking.sol

Here we create a new character by shifting variables. This concept can be though as a 'virtualstruct'. Essentially instead of creating a 'struct' as Solidity expects it and let the compiler do the parsing, we do it ourselves. That way, we achieve gas costs which are substantially lower.

Table 4.2: Gas costs for Byte masking method without Library

Optimizer Runs	Register	CreateCharacter	Constructor
0	70003	66620	551800.0
1	69943	66365	378022.0
100	69811	65924	402120.0
500	69604	65855	419559.0
500000	69598	65855	432537.0

:

In addition, as we essentially do the optimization ourselves, the deployed bytecode is smaller. This is not exactly intuitive, as it'd be expected that the solidity compiler is able to pack everything perfectly. It turns out⁵ that the compiler is not very efficient and as a result this method is far more efficient. With this method we are able to store and fetch all the data in a very efficient way, which costs X% gas less than the previous implementation. However, this method does not allow for a readable and maintainable interface. In order to export every functionality it is needed to convert the 'uint' variables to bytes to perform the bit operations on functions. This creates undesired overhead and thus is avoided.

GameByteMaskingLib.sol

It is important to consider code reusability, in the case another developer wanted to develop on the same structure, they should not need to deploy the core functionality of the 'Character' structure each time. Utilizing the 'using X for Y' syntax, we can export the library's API in a format that is similar to calling functions on struct's in Golang⁶.

There are two ways to export functions when creating Solidity APIs:

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

⁵<https://github.com/figs999/Ethereum/blob/master/Solc.aComedyInOneAct>

⁶golangtutorials.blogspot.de/2011/06/methods-on-structs.html

2. Public: This is a more complex process:

- (a) The library contract gets compiled and deployed
- (b) The main contract gets compiled and has placeholder slots in the bytecode.
- (c) The placeholder gets replaced by the deployed library's address
- (d) Any function call that requires the library utilizes the 'delegatecall' opcode.

The usage of the former can be done for separating the code and creating a more well done repository. The latter's usability can be seen with more general purpose functions such as error-checked functions for mathematic operations. That way, instead of everyone having to deploy their own version, they can use the already deployed one. The tradeoff comes between deployment costs and calling each function. When the bytecode is inlined, the jumping is done internally, while delegatecall requires additional resources. Finally there is a security gain, such as when everyone uses the same version of an audited library compared to everyone deploying their own. We opt for the internal approach, because cheaper and does not make sense to deploy, i.e. not enough people will care about it.

The final version is split in two files, a library file and a main file. [EXPLAIN LIBRARIES].

4.3.3 Results

It can be seen that in all cases the optimizer's first iteration creates significant gas savings. However, the more optimizer-runs were done, the more gas cost was spent during deployment, however the cost of 'CreateCharacter' went down. Code in Solidity is either optimized for size, and thus costs less to deploy, or for runtime costs, which costs more to deploy but each function costs less [1].

We described a technique which relies on the compiler's optimizer to pack the data in a struct and do the gas savings, however is simpler and more elegant. The second and third technique are more complex and allow for further gas optimizations. The second technique is more efficient however lacks reusability and is less maintainable. On the other hand, utilizing libraries however we can export a user-friendly API for reusing our code for anyone who has the same use case as us. This technique will be utilized in the Design and implementation section.

5 Smart Contract Security

Smart contracts immutable, lack of tooling, and developer mistakes

5.1 Past Vulnerabilities

Past vulnerabilities had lots of money locked, business logic of application broken. Brief summary of some vulns. Refer to relevant literature for more in depth.

Online platforms for training and exploiting have been developed.

5.2 Security of deployed Smart Contracts

Talk about the literature which has already mapped through the deployed contracts. Which tools did they use? Oyente, Mythril etc

5.3 Best Practices

Contract oriented, keep it simple, no tx.origin, ktl as described in literature

5.4 Access Control

There is NO private data, just functions that can be called by certain individuals A proper access control model needs to be implemented so that only authorized users can access certain functions.

6 Blockchain and the Energy Market

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

List relevant projects in energy sector

6.1 Advantages of Blockchain

Transparency, full history of meter readings, price calculation, billing of inhouse energy departments. This can be extended for EV car payment microtransactions and so on.

6.2 Our Use-case

Describe meters, billing and so on

7 Design and Implementation

7.1 Business Logic

Explain company structure

7.2 Smart Contracts

Explain the Smart Contracts suite

7.3 Monitoring Server

Explain monitoring server

7.3.1 REST API

Explain rest api usage

7.3.2 Python Client

Explain python implementation of rest api

7.3.3 web3.py interaction

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

8 Conclusion

8.1 Results

8.2 Future Work

Final remarks include.

Bibliography

- [1] Optimizer seems to produce larger bytecode when run longer. <https://github.com/ethereum/solidity/issues/2245>.
- [2] 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/.
- [3] Colored coins, 2013.
- [4] Gas costs from yellow paper – eip-150 revision (1e18248 - 2017-04-12). https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWc0lRem_m009GtSKEKrAsf07Frgx18pNU/edit#gid=0, 2017.
- [5] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.
- [6] Vitalik Buterin. On public and private blockchains. <https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains>, 2015.
- [7] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
- [8] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. *CoRR*, abs/1703.03994, 2017.
- [9] Digiconomist. Bitcoin energy consumption index. <https://digiconomist.net/bitcoin-energy-consumption>.
- [10] Ethereum. Ethash. <https://github.com/ethereum/wiki/wiki/Ethash>.
- [11] Ethereum. Recursive length prefix (rlp). <https://github.com/ethereum/wiki/wiki/RLP>.
- [12] Jorge Izquierdo. Library driven development in solidity. <https://blog.aragon.one/library-driven-development-in-solidity-2bebcaf88736>, 2017.
- [13] Preethi Kasireddy. How does ethereum work, anyway? 2017.

- [14] Florian Tramèr Lorenz Breidenbach, Phil Daian. Tokenize gas on ethereum with gastoken. <https://gastoken.io>, 2018.
- [15] J.P Morgan. A permissioned implementation of ethereum supporting data privacy. <https://www.jpmorgan.com/country/DE/en/Quorum>.
- [16] Donald R. Morrison. Patricia&mdashpractical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.
- [17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- [18] Chance Santana-Wees. Virtualstruct.sol. <https://github.com/figs999/Ethereum/blob/master/VirtualStruct.sol>.
- [19] Nick Szabo. Smart contracts: Building blocks for digital markets, 1995.
- [20] Parity Technologies. Proof-of-authority chains. <https://wiki.parity.io/Proof-of-Authority-Chains.html>.
- [21] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.