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

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 deparments 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 [14]. 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 transfering value from A to B. The concept of colored coins, [1] was introduced, where users were able to embed extra data on a bitcoin which resulted in coins that could represent represent ownership over a land title or a domain name. In 2015, Vitalik Buterin authored the Ethereum Whitepaper [5] 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 [16] 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 furter analysis.

2.4 Outline

Chapter X describes Y

3 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 cryptograpy 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 [18].

3.1.1 Public Key Cryptography

Also refered to as Assymetric Cryptography, it is a system that uses a pair of keys to encrypt and decrypt data. The two keys are usually called **public** and **private**, 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 enrypt 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 fullfil certain security properties and are used in cryptograpy

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:

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

```
  r: '0
      xe7df9fee346ea3575dd541e120df7b154d46bba2bf859352b29a7f2c6dd5fab3
      ',
  s: '0
      x68d266b140ac468ee328592ca6b4a461edace859d6837358bc3f8eeeb24ade63
      '
}
```

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.

```
> web3.eth.getBlock(5284738)
{ difficulty: BigNumber { s: 1, e: 15, c: [
      32,
      85319757566868
    ]
  },
  extraData: '0x7869786978697869',
  gasLimit: 7995219,
  gasUsed: 1547361,
  hash: '0
      x61ff0118470fdda14815bdc26f6e4fb29effc55369f3d6985e1433f782686403
      ',
  logsBloom: '0
      x0002080000020400020004000000000000010040000000000800020000000084000800400220
      ',
  miner: '0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb',
  mixHash: '0
      x29b6efa55ad0298b0c90f21e9e23d572977ffb3c5064a9816a69bb2bf2a9effd
      ',
  nonce: '0xabed128000fed25e',
  number: 5284738,
  parentHash: '0
      xb7063b9c7b05c95c35a329717e44875829cc740b2e0749e03d54806dcf34b520
      ',
  receiptsRoot: '0
      xe5e176557b9f40394917191095b706a2a331742f0dc93a10e1d59b5e297ee0b5
      ',
  sha3Uncles: '0
      x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
      ',
  size: 7789,
```

```
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 netwokr. This is one of the main benefits of using a blockchain, transparency.

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

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

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

Figure 3.1: Blockchain = chain of blocks

Vitalik Buterin goes indepth in the advantages and disadvantages between private and public blockchains in [4]. 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 [12], 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 [13] 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 [9] of the account's data.

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

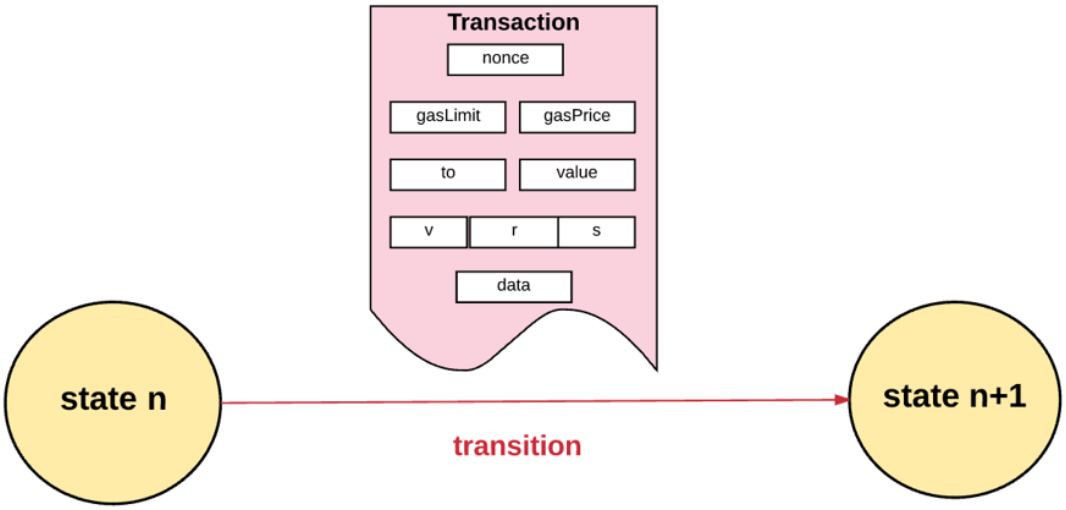

Figure 3.2: After each transaction the state gets updated

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 transfering 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 [2, 18].

[INSERT GAS / WEI / ETHER denominations]

When refering 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:

$$totalTransactionCost = gasPrice * gasLimit \tag{3.1}$$

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.

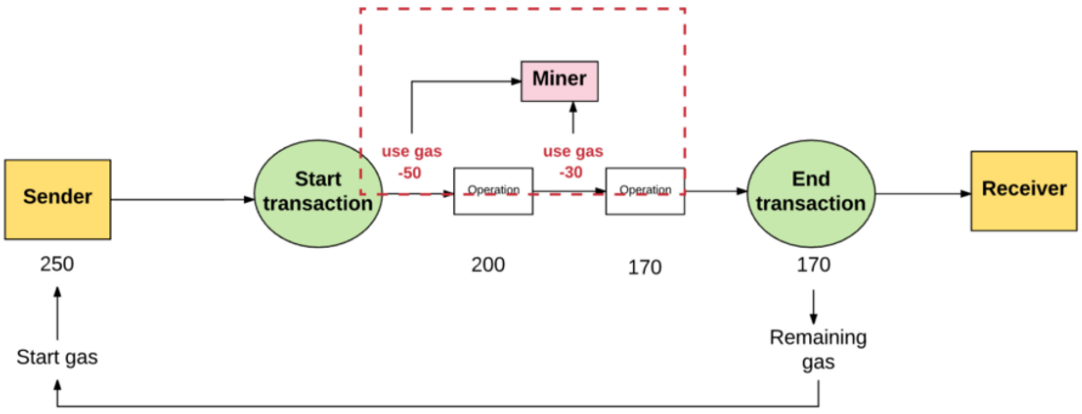

Figure 3.3: Successful transaction (from [10])

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

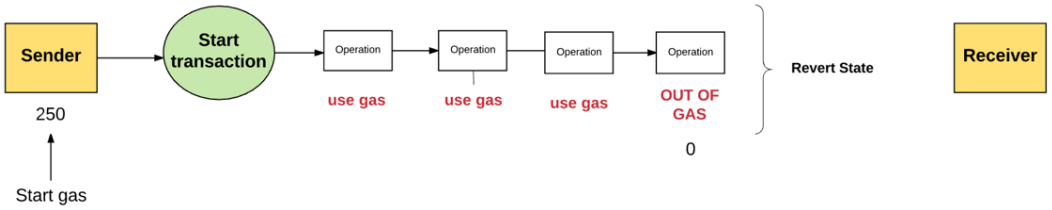

Figure 3.4: Out of gas transaction (from [10])

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 occured due to them providing programmers with powerful state-changing functions,

```
pragma solidity ^0.4.16;

contract TestContract {

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

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

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

Figure 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 [17] consensus algorithm, compared to Ropsten running the Proof of Work [8] 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 ganache[1] (formerly known as testrpc).

Web3

Web3 is the library used for interacting with an Ethereum node. The most feature-rich implementation is Web3.js[2] 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[3]. We showcase an example of connecting and fetching the latest block from Ropsten and Mainnet using Web3.js and Web3.py. The full specifications of each library's API can be found in their documentation[4][5]

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

```
$ 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
```

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 avereage 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 bandwith 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 miners[1].

Long term solutions in Ethereum are categorized in:

1. Sidechains: First described in [3], 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[7]. 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 blokcchain. The proposed technique involves exchanging signed messages offchain 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 [6], 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.

[INSERT TABLE ON GAS COSTS SHOWING SSTORE ETC]

4.3.1 Gas Costs

A smart contract gets compiled to EVM Assembly. There, by inspection or by using tools we can estimate the gas cost for deploying a contract or calling a function. The EVM operates on 32 byte words and requires a SSTORE command every 32 bytes. It also applies packing, which means that 2 128 storage variables can be stored with 1 SSTORE command. The *optimize* flag of the Solidity compiler needs to be activated to access this feature when programming in Solidity.

```
pragma solidity ^0.4.21;

contract Packing {

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

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

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

Figure 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 Optimizing for gas

Various methods have been proposed for saving gas costs. A recent construction[11] describes pre-buying gas at low cost periods in order to spend when prices are higher at times when the network is congested.

General rules that should be followed for saving gas costs:

1. There is no reason in saving chunks of data on a blockchain, unless the data itself needs to be stored forever. If the data's proof of existence needs to be stored, a 32 byte hash is sufficient.

2. Enable compiler optimizations

3. Use 'delete' on unused variables. SSTORE's formula refunds when setting back to 0. [REPHRASE]

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. FACTCHECK

5. ADDMORE

Finally, as described in [6] there is a lot of room for further compiler optimizations. Future Solidity compiler versions are addressing some already[2][3][4]

4.3.3 Encoding

We describe a technique of bitmasking and shifting data, influenced by [15]. It allows for gas savings compared to when storing as a normal structure. This compromises some of the readability of the library, however we can export a user-friendly API which allows to save up to X gas on data storage. This will be utilized in the implementation in Section X.

CONTINUEHERE

[2] https://github.com/ethereum/solidity/issues/3760
[3] https://github.com/ethereum/solidity/issues/3716
[4] https://github.com/ethereum/solidity/issues/3691

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

```
pragma solidity ^0.4.16;

contract TestContract {

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

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

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

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] Colored coins, 2013.

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

[3] 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.

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

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

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

[7] Digiconomist. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption.

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

[9] Ethereum. Recursive length prefix (rlp). https://github.com/ethereum/wiki/wiki/RLP.

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

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

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

[13] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

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

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

[16] Nick Szabo. Smart contracts: Building blocks for digital markets, 1995.

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

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