FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis in Computer Science

Sarah Maria Hyatt

Behavioural Analytics Framework for the Ethereum Blockchain

Department of Computer Science,
University of Copenhagen

BSc Supervision by Boris Düdder

June 4, 2019

Contents

1. Introduction

Blockchain technology (Peck, 2017) has been given a lot of attention the past years. While it emerged with Bitcoin in 2009 (Nakamoto, 2009) it has been a generator of research for domains such as cloud computing, security, and of practical importance as an emerging technology in financial and legal domains as well as medical supply chain industries (Amani, Bégel, Bortin, & Staples, 2018).

A blockchain, also known as a distributed ledger, is a private or public decentralized, append-only data structure, containing nodes that share a global state. A block contains multiple transactions and the nodes on the blockchain agree on the time order in which the blocks are processed. A blockchain can, therefore, be viewed as a log of ordered transactions (Dinh et al., 2017). One of the cornerstones of blockchains is that they do not rely on a trusted central authority. Each transaction is processed by a large network of mutually untrusted peers, by the name of miners (Nicola Atzei & Cimoli, 2017). Blockchains can meet the expectations of this trait due to a consensus protocol, typically a mathematical computation performed by the network, to verify transactions.

Ethereum (Wood, 2014), initially released July 2015, is currently one of the dominant blockchain systems, having a current market capitalisation of more than 26[1] billion US dollars (*Ethereum Market Cap*, n.d.). Like Bitcoin, Ethereum offers a cryptocurrency, which is a digital asset, called Ether. However, Ethereum should not only be seen as a cryptocurrency as it is also an environment.

Ethereum provides the freedom to create Turing-complete programs, predominantly written in the Ethereum programming language Solidity (Bragagnolo, Rocha, Denker, & Ducasse, 2018). These programs are smart contracts. A smart contract is a set of functions that encode and enforce agreements (Szabo, 1997), which, in other words, is a computer program. It is called smart because it can combine protocols with user interfaces to formulate and manage communication across computer networks, and it is called a contract because it is cryptographically signed by a private key belonging to the account invoking it. Ethereum is an environment because it lets users create decentralized applications (DAPPs) that build upon the blockchain technology provided by Ethereum (Buterin, 2013), as well as creating arbitrary computational programs that perform transactions and Ether transfers (Nicola Atzei & Cimoli, 2017). A shareholder for a DAPP would be defined as a work token whereas a cryptocurrency for a DAPP would be defined as a usage token.

Smart contracts are executed in the form of Ethereum Virtual Machine

[1]Retrieved: 21/05/2019 2:00 PM

(EVM) bytecode (Bhargavan et al., 2016). Successfully creating a smart contract creates a contract account which can be identified by its unique contract address.

Figure 1 demonstrates an abstract overview of how the component of Ethereum and how smart contracts interact. The users are EOAs or owners, who can create smart contracts. These are contract accounts containing their own private storage, a state, a balance holding some amount of Ether and executable code, their contract. Smart contracts are deployed through transactions. When a range of transactions is ready for deployment the miners compete on being the quickest to perform the consensus protocol. The quickest miner gets to deploy all the transactions into one collected block and append it to the blockchain. In the mining process, the miners collectively agree on the current state and transactions in order to determine the next state of Ethereum.

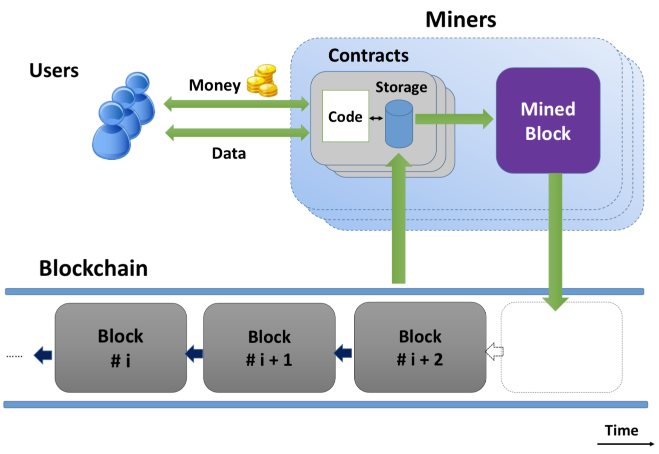


Figure 1. How the components of Ethereum work together, the figure is from (Delmolino et al., 2015).

Once a transaction is executed to the blockchain it cannot be reversed, and since Ethereum is a public blockchain, the transaction is publicly visible. Each contract account's state is adjusted by its private storage and the amount of Ether it holds (Luu, Chu, Olickel, Saxena, & Hobor, 2016). If contracts exist within the global scope of Ethereum's state they can communicate together through messages or internal transactions, this will be elaborated in 2. Background. The order of transactions determines the state of each contract, thus controlling the balance of each user (Nicola Atzei & Cimoli, 2017). So generally speaking all states on Ethereum are updated after every transaction, meaning that Ethereum can be seen as a transaction-based state machine (Wöhrer & Zdun, 2018). Once a contract is deployed to the blockchain, debugging becomes very difficult. The contract cannot be re-executed and it is encoded in EVM bytecode (Bragagnolo et al., 2018).

The intention of this thesis is to develop a research framework where the focal point is to investigate the communication between contracts as well as discovering whether patterns occur of the relationship among contracts and owners. The rationale is that one can observe that some contracts are highly replicated on Ethereum, for instance many copies of some contracts, although the reasons that prompted these replications are not apparent. The computational power of smart contracts, being Turing-complete programs, raises interest to investigate the communication between contracts to possibly identifying patterns. Thus the implementation of a logical contract may consist of several smart contracts interoperating together, creating cycles or simply large chains of transfers. Analysing the patterns could help uncover potential scams such as pyramid schemes.

However, without an automatic tool, it is tedious to identify and analyse such odd-behaving contracts and patterns in owner-contract relationships. This motivates the approach of this thesis that aims to

1. cluster contracts in regards to ownership and contract replications, and
2. identify communication patterns between contracts.

Thus there was a significant set of technological challenges, which in order to solve, I had to engineer a practical solution. One of the main challenges was meeting a performance where the trade off was high performance as opposed to low storage usage. The final performance for the preprocessing step was 2,6 hours, being a speed-up of factor 31. The performance of the clustering process was 4,4 hours. The technological challenges are elaborated further in 5.5 Difficulties.

1.1 Related Work

Various approaches to related work focus on individual smart contracts. Several studies (Wöhrer & Zdun, 2018) survey typical pitfalls for Solidity smart contract developers. Here they have designed a range of design patterns that target specific security issues such as re-entrancy attacks and patterns that involve emergency stops to prevent malicious contracts of being executed, among others.

Other studies (Bragagnolo et al., 2018) that focus on individual smart contracts where they have created a tool that can reflect deployed objects on what is essentially a reflection-less system, like Ethereum where everything is stored as bytecode. They call it SmartInspect. The purpose of SmartInspect is to visualise the contract code and to allow debugging without having to redeploy the contract, nor develop any ad-hoc code (Bragagnolo et al., 2018).

Additional studies approach smart contracts differently. (Rocha, Ducasse, Denker, & Lecerf, 2017) have used Smalltalk Compiler-Compiler (SmaCC) (Brant, Lecerf, Goubier, Ducasse, & Black, 2017) to implement a parser for Solidity contracts that works with Pharo (Bergel, Cassou, Ducasse, & Laval, 2013). Their paper takes the reader through the process of developing their parser along with the irregularities of the Solidity language occurring during the process. The process of discovering these irregularities in addition to ambiguities makes their parsing tool useful to help understand debugging and security bugs in smart contracts.

1.2 Contribution

Instead of investigating individual smart contracts this thesis looks at the blockchain as a whole and studies the dependencies between smart contracts. This thesis demonstrates the usefulness of such a tool by identifying smart contracts that are susceptible to build pyramid schemes or the likes and learning the mechanisms of how contracts are replicated.

- Analysing contract accounts with respects to their owners.
- Investigating the occurrence of replicated contracts.
- Developing a tool to demonstrate patterns in transactions between contracts.

This thesis is a Bachelor of Science done at the Department of Computer Science, University of Copenhagen under the supervision of Boris Düdder in the period February 5, 2019 till June 11, 2019.

1.3 Roadmap

The remainder of this thesis is organised as follows. Section II will give a background of Ethereum, transactions, smart contracts and basic concepts of extracting the Ethereum blockchain. Section III analyses the requirements and the primary design of the application domain. Section IIII demonstrates the architecture of the framework, in addition to the design of the execution process. Section V provides a technical discussion of which tools this thesis applies. Section VI gives a brief description of how to run the framework. Section VII presents an alternative solution as to preprocessing the Ethereum transaction data. Section VIII demonstrates the results of clustering contracts with respect to their owners, whilst presenting additional findings and providing an evaluation of the work. Finally, Section VIIII presents conclusions and outlines potential future work.

1.4 Table Summarising the Components

Ethereum	A blockchain system with a cryptocurrency and an environment to develop on top.
Ether	The cryptocurrency of Ethereum.
EOA / owner	Accounts the are owned by people, which do not contain any code.
Contract account	Accounts created by EOAs, containing a contract of code that controls them.
Smart contract	The contract of code attach to a contract account, with a set of functions that encode and enforce agreements.
DAPP	A decentralised application created on Ethereum's environment.
Work tokens	Identifies as a shareholder of a DAPP.
Usage tokens	Acts as a cryptocurrency for its respective DAPP.
Miner	Competes to solve a puzzle to deploy a block of transactions to Ethereum.
Invokes tree	A database resembling a call graph, showing contract-to-contract transfers.
CC tree	Tree-structured database to group contracts with respect to their EOAs.
Ether transfers	EOA to EOA transactions or EOA to contract transactions.
Contract-to-contract invokes	Transactions between contracts that are not tokens.
Contract Creations	Transactions that involve creating new contract accounts.

2. Background

The following section will provide some brief background knowledge of the components relevant for this thesis.

2.1 Contracts and Owners

Ethereum uses two types of accounts, externally owned accounts (EOAs) and contract accounts. Each account has a 20-byte address and state transitions which are direct transfers of value and information between accounts (Buterin, 2013). The main difference between EOAs and contract accounts is that and EOA is controlled by its private key and does not have contract code attached to its account, while contract accounts are controlled by their contract code. EOAs can be thought of as people that own accounts like they would a bank account. Although contract accounts are created by EOAs, they exist independently on the blockchain, meaning that any EOA can make a call to that contract or create a contract that makes a call to it (Wöhrer & Zdun, 2018).

Contract accounts can only invoke transactions upon other transactions it has received, in this thesis, it is referred to as contract-to-contract invokes. Contract-to-contract invokes can lead to various actions on the blockchain, since it is possible to create contracts that invoke multiple different contracts on the blockchain. These invoke can be both reads, writes, interacting or executing other contracts and transferring values.

Since smart contracts are Turing-complete programs, it is possible to implement all sorts of features in each contract. One possibility is a contract creating one or more contracts in its contract code, another possibility is a single contract creating transfers to multiple other contracts.

```
 1  pragma solidity ^0.4.24;
 2
 3  contract Smartest {
 4      mapping (address => uint256) invested;
 5      mapping (address => uint256) investBlock;
 6
 7      function () external payable {
 8          if (invested[msg.sender] != 0) {
 9              msg.sender.transfer(invested[msg.sender] * (block.number -
                    investBlock[msg.sender]) * 21 / 2950000);
10          }
11
12          investBlock[msg.sender] = block.number;
13          invested[msg.sender]    += msg.value;
14      }
15  }
```

Figure 2. An example of a smart contract deployed on Ethereum's blockchain.

Figure 2 is an example of a smart contract, written in Solidity and deployed on Ethereum's blockchain. Line 1 states the version of Solidity, which is very important, since the Ethereum platform is increasingly extended and modified, for improvements, bug fixes et.c. Solidity developers have to consider these constant changes - the code they create today might not compile in a few months (Wöhrer & Zdun, 2018). The contract *Smartest* begins on line 3.

Lines 3-4 are mappings of the addresses, where `invested` stores the address of the sender and the invested amount. `investBlock` stores the address of the sender and the block number of the last investment. `msg.sender` and `block.number` are global variables that do not have be stated within the contract. `msg.sender` is the address currently interacting with the contract, which can be both another contract or an EOA. `block.number` refers to the latest block number on the chain.

The nameless function with no parameters, beginning on line 7, is a fallback function (*Solidity*, n.d.). A fallback function is a default function that is executed on a call to the contract if no other functions match the given function identifier. `payable` makes sure that the contract can receive Ether. `external` (*Solidity*, n.d.) is a modifier applied to the function to make it externally callable. A modifier is an inheritable property of contracts that may be overridden by derived contracts. They make is easy to change the behaviour of functions. Line 8 makes a check to see if the sender has made any investments if so, line 9 performs a computation of an investment payout for the sender. There are approximately 6000 new blocks added each day[2], which is why the computation of `(block.number - investBlock[msg.sender]) * 21 / 2950000)` refers to `6000 * 21 / 2950000`. Where `6000` is a period of approximately a day. The investment payout of this contract is then 4,3%, being the amount sent to the investor. Lines 12-13 then update the mappings of the investments and the block number of the last investment.

2.2 Transactions and Messages

The term *transaction* is used in Ethereum to refer to the signed data package that stores a message to be sent from an EOA. There are two types of transactions; *message calls* and *contract creations*. Transactions are set in motion by EOAs signing the transaction using their private key. Contract accounts communicate via messages or internal transactions, these internal transactions exist in the Ethereum execution environment and are generated in the contract code. Looking into a transaction, the sender will always be the EOA in charge of playing out the transaction and therefore always be the `from_address` of the transaction. Any EOA can activate a contract account's code and cause a transaction (*Ethereum Homestead*, n.d.).

[2]https://etherscan.io/chart/ethersupply

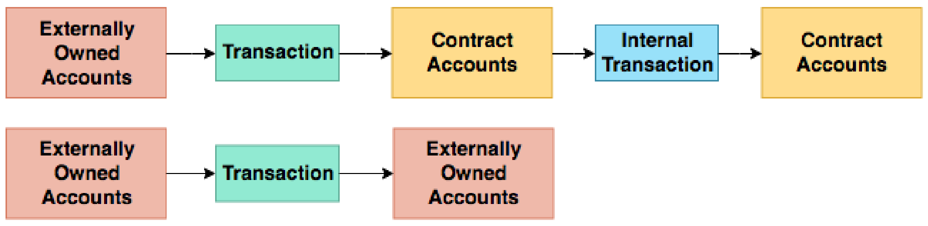


Figure 3. Example of a contract-to-contract invoke executed by an EOA and a transfer between two EOAs.

Figure 3 is an example of a contract-to-contract invoke executed by an EOA, and a transfer between two EOAs. For a contract-to-contract invoke there are typically functions specified in the input data of the transaction. Two types of these functions are `transfer` and `transferEvent` that specify a call being made to a contract in which a transfer event is occurring. In these cases, it is always a transfer event invoked by a contract and it is possible to extract the information of the receiver, who can be either an EOA or a contract. Finally, there are also contract creation transactions. These differ by the rest because they do not have a `to_address`, meaning a receiver. Instead, they are given a unique contract account address along with their smart contract attached to the contract account address.

2.2.1 Tokens

As mentioned in the previous section, Ethereum has an environment to create DAPPs (Prasad, Dantu, Paul, Mears, & Morozov, 2018). The usage tokens act as a currency and can represent particular assets or utilities. Both tokens are controlled by smart contracts. ERC20 is currently the most applied type of token[3], having a specific set of functions that can be applied when interacting on a DAPP.

These functions and general functionalities of token transactions are like those deployed directly on Ethereum. However, tokens have the freedom to specify additional functionalities.

[3]https://theethereum.wiki/w/index.php/ERC20_Token_Standard

2.2.2 Gas

In order to perform a transaction, one must pay an operational gas determined by the computational resources in use (Buterin, 2013). The higher the resources required, the higher the cost of gas. The purpose of gas is to avoid exponential blow-ups and infinite loops in code, potentially causing denial-of-service attacks[4], where users try to overrun the network with time-consuming or infinite computations (Wöhrer & Zdun, 2018). `STARTGAS` is the limit set to prevent these actions, and `GASPRICE` is the expense to pay the miner for each computational step.

In cases where the limit is reached and the transaction execution is not completed, the state changes will revert and a status in the transaction called `receipt_status` will be marked as 0, meaning a failure occurred in the transaction. However, the payment of the expenses will not revert and the miner is paid for the computation performed up to that point. If a transaction execution halts with some gas remaining, then the remaining portion of the fees is refunded to the sender (Buterin, 2013).

2.3 The State of Ethereum

Viewing Ethereum as a transaction-based state machine, the initial state is what is defined as the *genesis state*. To begin with, the genesis state is also the final state. When transactions are executed, a state is added as the new final state. The final state will always be the current state of Ethereum. The state of Ethereum has millions of transactions that are grouped into *blocks*. The blocks are chained together, forming a blockchain, where each block contains a range of transactions.

[4]A denial-of-service attack (DoS attack) is an attack where the perpetrator makes the machine or network resources unavailable, typically by flooding the targeted machine's resources with excessive requests with the purpose of overloading the system.

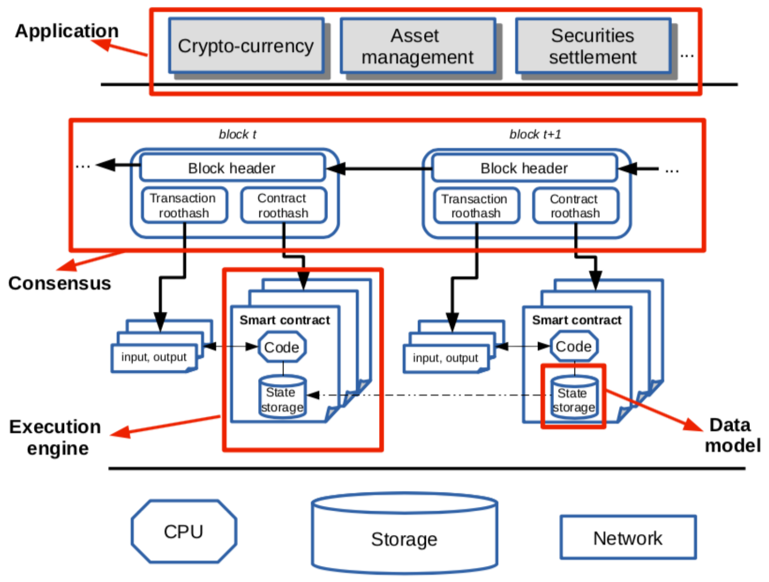


Figure 4. Ethereum blockchain software stack (Dinh et al., 2017).

Figure 4 illustrates the Ethereum blockchain software stack, where the transactions are added to blocks that are linked to the previous, creating a chain. The order of transactions determines the state of each contract, thus controlling the balance of each user (Nicola Atzei & Cimoli, 2017). However, it is the miners who decide the order of transactions in the block they mine (Luu et al., 2016).

2.3.1 Mining

Mining is a validation process of transactions. Nodes (i.e. computers) go together in groups and expand their compute resources to create a block of valid transactions. The miners perform a mathematical proof, known as *proof of work*, where the fastest miner will be rewarded. Every time a miner proves a block, new Ether tokens are generated and awarded. Once the validation process is completed, Ethereum will move into a new final state.

2.4 Web3 API

Figure 5 is a visualisation of the Web3 API connected to Ethereum. One can use a API to connect to Web3, an example of such available API services is Infura (*Infura*, n.d.). The Web3 API makes it possible to interact directly with the live blockchain, facilitating the latest information from the blockchain through network. The API receives transfers, contract creations and contract transactions (function calls). The Core component signs the transaction with the senders private key, after which it is sent to the Network component where the process of mining enacts. Once the process of mining is completed, the transaction is stored to the latest block in the blockchain and a transaction receipt is returned to the sender.

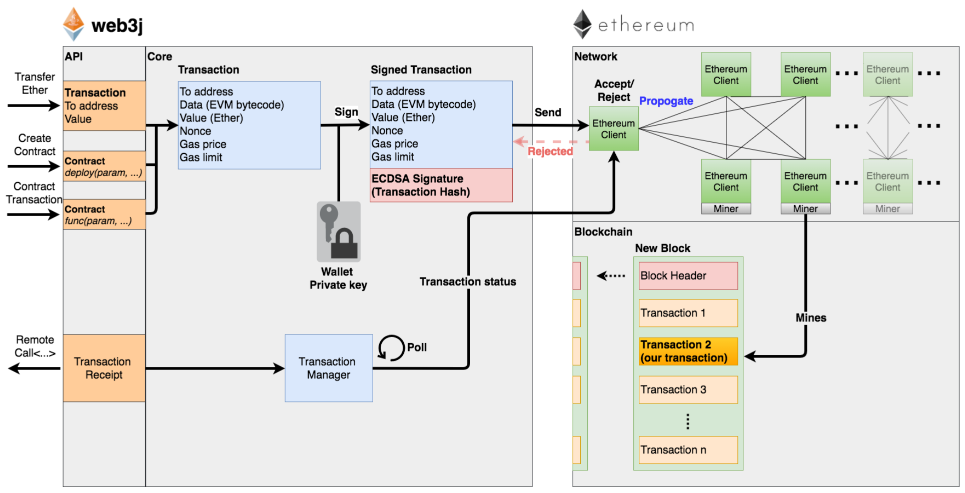


Figure 5. Web3 API and the Ethereum Blockchain (*Web3*, n.d.).

The `web3-eth` package is useful to interact with Ethereum and the deployed smart contracts. A few examples of the possible applications of extracting data from Ethereum using Web3 are:

```
web3.eth.getCode()
```

Figure 6. Gets the code of a contract or EOA.

```
web3.eth.getTransactionCount()
```

Figure 7. Gets the nonce of an EOA.

```
web3.eth.getBlock
```

Figure 8. Get a block.

`web3.eth.blockNumber` outputs latest block number, while `web3.eth.getBlock` returns the block specified by its number. Combining the two it is possible to get, for instance, the latest ten blocks as the code below illustrates.

```python
for i in range(10):
  block = web3.eth.getBlock(web3.eth.blockNumber)
  print(block-i)
```

Figure 9. Prints the latest ten blocks on Ethereum.

2.5 Google BigQuery

Google BigQuery offers high performance and scalability for data analytics systems, in addition to data that users can use to understand how the system works (Fernandes & Bernardino, 2015). The full Ethereum Blockchain is provided as table data that one can execute high-performance querying using SQL[5]. Extracting parts of the blockchain can be done with Pandas, where BigQuery's provides an integrated library. In addition, BigQuery provide table data available to export as `Avro`, `CSV.gzip` and `json.gzip` files (*Exporting table data from Big Query*, n.d.). The fully `json` gzipped version of the Ethereum blockchain takes up more than 176 *GB* space. The chain data is already presorted into partial sections: `contracts`, `transactions`, `logs`, `blocks`, `token-transfers`, `traces` and `tokens`, where each section contains a range of files. The entire blockchain can be exported within a relative amount of time. Below is an example of one transaction withdrawn from BigQuery's Ethereum data transactions files.

```
1.    {'hash': '0xecfee04e3c04490bdac99ea00b75bd620f7a36e9f6b9919d47e01376644233bd',
2.    'nonce': '0',
3.    'transaction_index': '118',
4.    'from_address': '0x91d0893509f3bd4271b2106f94de51dfd459edea',
5.    'to_address': '0x0bd57fc1289c2b8e090e5f2f7d972514e6a494fb',
6.    'value': 0,
7.    'gas': '57016',
8.    'gas_price': '18000000000',
9.    'input': '0xa9059cbb0000000000000000000000000956ead78bd38d39d486a058a22cf35334
10.             1fc3a9000000000000000000000000000000000000000000000021e19e0c9bab2400000',
11.   'receipt_cumulative_gas_used': '5461853',
12.   'receipt_gas_used': '22016',
13.   'receipt_contract_address': None
14.   'receipt_status': '1',
15.   'block_timestamp': '2019-02-25 09:14:30 UTC',
16.   'block_number': '7265339',
17.   'block_hash': '0xd667a2d324f8122f4f5f15bd1669a68ba3a519150c648d4b1df72e4236efbb69'}
```

Figure 10. A single example of a transaction from BigQuery's Ethereum transaction data.

The above shows an example of a transfer event due to `0xa9059cbb` being the first ten characters of the `input` data on lines 9-10. The input data is where bytecode would be stored. As in this example, it refers to a function call, in a contract creation the code of a contract would be stored here and if it were a message being sent, the message would be stored here as well. The example is not a contract creation because a `to_address` is given on

[5]https://bigquery.cloud.google.com/table/bigquery-public-data:ethereum_blockchain.transactions?pli=1

line 5 and we are not given a `receipt_contract_address` on line 13. The `nonce` represents the number of times a transaction has been performed, it is counted incrementally to make sure each transaction can only be processed once (Buterin, 2013). The above nonce on line 2 shows that this is the first time this transaction has been executed. A contract account can also have a nonce. However, a contract nonce will refer to the number of contracts creations that a contract account has performed. The contract nonce is visible in the contract creation transactions.

As for the rest of the transaction details in the above, the description of each is as follows.

`hash`	Refers to the unique hash string of the transaction.
`transaction_index`	Is an integer representing the transactions index.
`from_address`	Address of the sender.
`value`	The value being transferred in Wei.
`gas`	Gas provided by the sender.
`gas_price`	Gas price provided by the sender in Wei.
`receipt_cumulative_gas_used`	The total sum of gas used by the time of execution in the block.
`receipt_gas_used`	The amount of gas used by this specific transaction alone.
`receipt_root`	32 bytes of post-transaction stateroot (pre Byzantium).
`receipt_status`	Either 1 (success) or 0 (failure) (post Byzantium).
`block_timestamp`	Timestamp of the block where this transaction was in.
`block_number`	Block number where this transaction was in.
`block_hash`	Hash of the block where this transaction was in.

2.6 Memory Hierarchy and Big Data

Due to Ethereum's increasing popularity, the size of the entire blockchain is more than 500 *GB* and in constant development. Considerations in regards to managing large data sets are therefore necessary. To gain a decent performance, it is worth understanding how the memory hierarchy in Figure 10 works. The further down the data is stored in the memory hierarchy, the longer it takes for the program to reach. The bottom *L6* refers to remote secondary storage, such as Web Servers and distributed file systems. Extracting data from *L6* is therefore an approach with slow performance (Bryant & O'Hallaron, 2011). *L2* through *L4* refer to RAM storage, which is the preferred access to data in order to get a high performance.

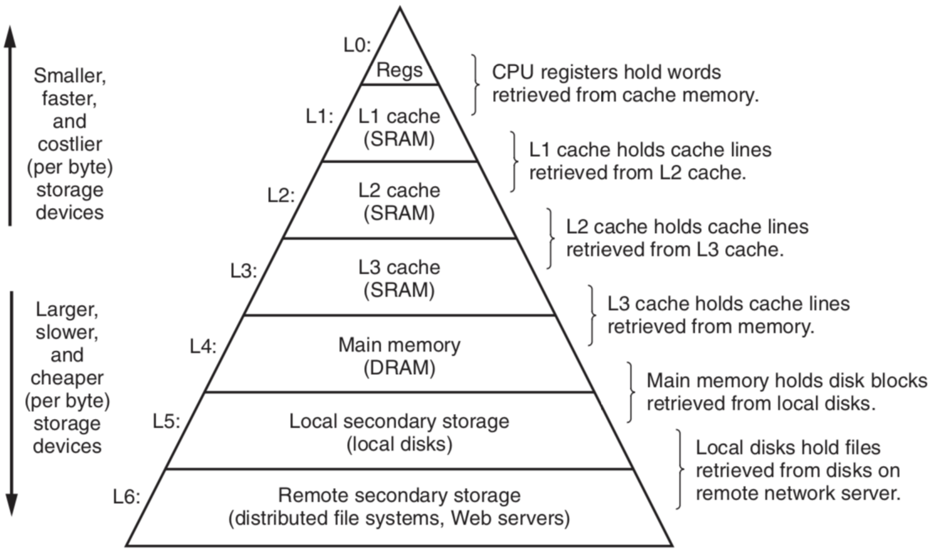

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network server.

Figure 11. Illustration of the memory hierarchy, (Bryant & O'Hallaron, 2011).

3. Analysis

This section dives into various options to solve the task at hand by analysing the requirements and matching approaches. Furthermore, this section gives a brief walk-through of the design specifications for data storage.

3.1 Requirements

One of the main goals is to achieve Ethereum transactions sorted against four overall types; contract creations to achieve the CC tree representing the contract relationship with respect to owners; contract-to-contract invokes to get the Invokes tree; Ether transfers corresponding to the additional transactions and all token transactions making it possible to investigate the differences of patterns between tokens transactions and transactions directly deployed on Ethereum.

To get a grouping of contracts with respect to their owner, the information needed as attributes for the database are as follows.

- The owner address, corresponding to the EOA creating the contract.
- The contract address, corresponding to the newly created contract account identifier.
- The contract code, to compute a hash value that can be used to check identical copies.

- The nonce, to see how many contract creations each particular contract account has created.
- The value, for the future purpose of analysing the flow of assets.
- The time stamp, to understand the timeline of the contract creations.

For extracting contract-to-contract invokes, the information needed as attributes for the database are as follows.

- The owner address, which corresponds to the EOA responsible for firing the transaction.
- The contract address sender, corresponding to the contract that performs a function call to another contract.
- The address receiver, corresponding to either the contract being invoked by a function call from another contract or the EOA, if the contract transfers to an EOA.
- A check to understand if the receiver is a contract account or an EOA.
- The input data, which contains the bytecode with information about the function calls et.c..
- The nonce, to see how many transactions the EOA has made.
- The value sent between the contracts, for the future purpose of analysing the flow of assets.
- The time stamp, to understand the timeline of the transactions.

Understanding the Ether transfers, the information needed as attributes for the database are as follows.

- The owner address, which corresponds to the EOA responsible for firing the transaction.
- The receiver address, which could be a contract account or another EOA.
- A check you understand if the receiver is a contract account or an EOA.
- The nonce, to see how many transactions the EOA has made.
- The value, for future purposes of analysing the flow of assets.
- The time stamp, to understand the timeline of the transactions.

3.2 Meeting the Requirements

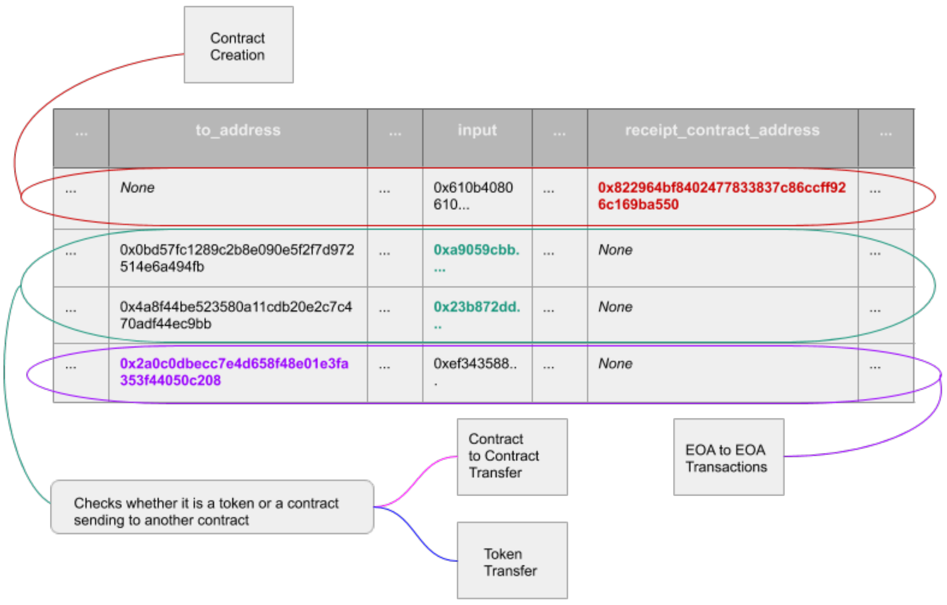
As described in the Background, this thesis explores two options for extracting the Ethereum blockchain, using BigQuery or Web3 API. This next section will discuss the advantages and disadvantages of both methods. A transaction contains the requirements stated in the previous subsection, it, therefore, makes sense to investigate the options of using transactions to gain the required knowledge.

3.2.1 Using BigQuery

BigQuery offers a range of methods to extract table data of the Ethereum blockchain. However, BigQuery's most interesting feature is the option to export the entire blockchain, as opposed to querying through the network or importing data through Pandas[6]. The following will explore the approach of exporting Ethereum as gzip compressed json files. Figure 12 shows an example of how a typical transactions file can be interpreted, from the BigQuery transactions data. The first row below, marked with red, shows a contract creation. This is due to the fact that it contains a `receipt_contract_address` and has no `to_address`. The `input` data of this type of transaction is the contract code.

The next two rows, marked with green, show contract transactions which are visible due to the function definitions of the `input` data. Input data that begins with either `0xa9059cbb` or `0x23b872dd` refers to a `transfer` and a `transferFrom` event. In addition, there is a `to_address` and no `receipt_contract_address`. It is important to note that these contract transactions can be either token contract transactions or contract-to-contract transactions deployed directly on the blockchain. Lastly, the bottom row, marked with purple, is an Ether transfer. It has a `to_address` which is typically an EOA address, it has no `receipt_contract_address` and the `input` does not contain any function calls.

[6]Pandas is a data analysis tools for the Python programming language, delivering high-performance, easy-to-use data structures for data sizes up to 1 GB.



...	to_address	...	input	...	receipt_contract_address	...
...	*None*	...	0x610b4080 610...	...	**0x822964bf8402477833837c86ccff92 6c169ba550**	...
...	0x0bd57fc1289c2b8e090e5f2f7d972 514e6a494fb	...	**0xa9059cbb. ...**	...	*None*	...
...	0x4a8f44be523580a11cdb20e2c7c4 70adf44ec9bb	...	**0x23b872dd. ..**	...	*None*	...
...	**0x2a0c0dbecc7e4d658f48e01e3fa 353f44050c208**	...	0xef343588.. .	...	*None*	...

Contract Creation
Contract to Contract Transfer
EOA to EOA Transactions
Checks whether it is a token or a contract sending to another contract
Token Transfer

Figure 12. An example of a transactions file from BigQuery and how it should be interpreted to extract the required data.

The Advantages of using BigQuery

1. It is a significant advantage to have the entire data of the blockchain stored in DRAM when creating our framework. The data will be visible, easily accessible and there is no risk that the data will change or become unavailable.
2. Processing the data runs fast since the data is in local storage rather than remote secondary storage.
3. BigQuery provides transactions files that specify the `receipt_contract_address`, making it easier to sort the contract creations.

The Disadvantages of using BigQuery

1. Requires a lot of storage because the entire blockchain is stored as a whole.
2. The BigQuery transactions files do not give knowledge of whether a transaction is a token or one directly deployed on Ethereum. Therefore this approach will need an extension that extracts all token contract addresses from the `token-transfers` files.
3. This approach is not immediately extendible in regards to gathering the latest Ethereum data. The cause is that the data is collected as a

whole, meaning it would need an extension to continuously download the latest block from Ethereum.

3.2.2 Using the Web3 API

Web3 offers a range of functions and additional libraries to operate on the Ethereum blockchain. Figures 13 and 14 demonstrate an example of how a transaction file is extracted from Ethereum.

```
from web3 import Web3

infura_url = "https://mainnet.infura.io/v3/859
    a0c3062564f2fa3fb978a3d465e77"
web3 = Web3(Web3.HTTPProvider(infura_url))

web3.eth.getBlock(web3.eth.blockNumber)
```

Figure 13. A small program that retrieves the latest block on Ethereum.

The output of these commands can be viewed in Appendix 10.1. From the output of the block one can extract all the transaction hashes and run them through the command below, which will return the transaction data.

```
web3.eth.getTransaction('0
    xda970ed3c54cc334b64344a07fe72ad58b6800510c3dbff8b82e01a5dc0e7972'
    )
```

Figure 14. Web3 command to extract a single transaction using the transaction hash.

Figure 15 is the output of the transaction call from Figure 14. The items have the same meaning of those described in 2.5 Google BigQuery, with the exception of r, s and v, that can be applied to generate the signature that identifies the sender of the transaction.

```
AttributeDict({'blockHash':
  HexBytes('0xbfdccc8e28916930e71ded1df813891ec16001b68026a082f7b946342d89c2b6'),
'blockNumber': 7880743,
'from': '0x4b4e1276A33FE48315084355723c8Aa8c84ADa5d',
'gas': 21000,
'gasPrice': 100000000000,
'hash':
  HexBytes('0xda970ed3c54cc334b64344a07fe72ad58b6800510c3dbff8b82e01a5dc0e7972'),
'input': '0x',
'nonce': 34829,
'r':
  HexBytes('0xcaa874fe823a9d288db4b034ddb3a9795e9fe4acbe477071c5eb972c505cbaf6'),
's':
  HexBytes('0x6784aedaf39cf0935d0866d422b0b956f9415c44e7298daacce67dc67dbb1e0c'),
'to': '0x19f6e297Ef2f5373e18D89d06B4947849A0cF58B',
'transactionIndex': 0,
'v': 38,
'value': 9623558340000000000})
```

Figure 15. Transaction data from running Web3 command.

The Advantages of using the Web3 API

1. Web3 provides easy access to the latest data, making it beneficial to use when constantly retrieving new data from the blockchain.
2. It fits well as a streaming approach where chunks of data are used and then discarded.

The Disadvantage of using the Web3 API

1. Working with a continuous stream on a network connection can often lead to slow performance and connection time outs.
2. Web3 needs an API, like Infura (*Infura*, n.d.), to connect to Ethereum. This draws a network passage as the one demonstrated in Figure 16. Each step in the network passage can lead to a slowdown in regards to performance because each step refers to the remote secondary storage.
3. Extracting transactions is a process of many steps, since first the block has to be extracted after which each transaction can be extracted.
4. Comparing the transaction results of Web3, in Figure 15, with those of 2.5 Google BigQuery, in the case of a contract creation, the address of the contract account is not stored when running Web3, as opposed to BigQuery. Meaning it would require additional computations to extract the contract account address.

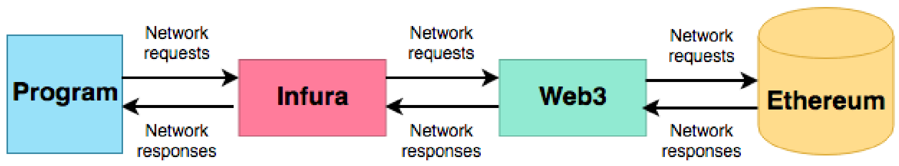


Figure 16. The network passage from program to Ethereum using Web3 and Infura.

3.3 Constraints of Big Data

Working with large data sets requires considerations in regards to the design of the program to get the best trade-off. A solution for increasing the performance of programs dealing with large datasets is applying multi-threading and/or multiprocessing. However, this does not stand alone. As explained in 2.6 Memory Hierarchy and Big Data accessing data stored in RAM is much more efficient than accessing data from remote secondary storages or even local secondary storages. If possible, it is if worth finding a solution that keeps access to the data in RAM. The trade-off gaining high performance is a high memory usage and vice versa. Albeit performance could also be increased with exploration of multi-threading and/or multiprocessing.

3.4 Choice of Approach

Working with a network connection, like Web3, would be beneficial if the overall size of data was either smaller or there was fewer data needed. Section 7 describes an experiment using the network and Web3, where an approach without network a connection gave a high speed-up.

For these reasons, going on this thesis will work with the BigQuery approach, exporting the entire blockchain, because performance and reliability are more sought after opposed to less storage.

3.5 Goals

The next section describes the goals to achieve in regards of creating CC tree and Invokes tree.

3.5.1 Clustering

Clustering is a data science technique of unsupervised learning used to interpret the behaviour of data from unlabelled data (Grus, 2015). KMeans is a clustering approach where the data is partitioned into k clusters in which each data point resides to the cluster with the closest mean.

Finding the optimal number of clusters can be tricky when working with unlabelled data. It is, therefore, useful to apply methods such as the Elbow Method (Grus, 2015). The Elbow Method is a useful tool for finding the optimal $\boldsymbol{k}$ because it plots the sum of squared errors, between each point and the mean of its cluster, as a function of $\boldsymbol{k}$. The best $\boldsymbol{k}$ is then interpreted by analysing the graph and finding the spot where the line bends (Grus, 2015).

Investigating the transactional behaviour concerning contract and owner (EOA) relationships, the following are goals to investigate applying to cluster.

1. Equivalence classes with owners that have the same number of contracts. In other words, if an owner has three contract accounts, how many EOAs do also have three contract accounts. The rationale is to

be able to study the common number of contracts in order to classify odd cases.

2. Comparing the number of unique contracts per EOA with the number of contracts per owner. In other words, investigating how many identical contracts an owner generally creates, if any. Some owners may appear to have created more than 100.000 contracts, which is why it is interesting to investigate whether these contracts are unique or replicated.

3. Comparing the number of times a contract occurs more than once with the number of unique EOAs using each contract. In other words, investigating how many identical contracts occur among different EOAs. This result should show the occurrences of identical copies across various owners, which could show a pattern for tendencies of developers copying other contracts.

3.5.2 Designing a Storage Model that Preserves the Structure

The goal is to construct a design where all contracts are grouped with respect to their owners. Part of this goal is to investigate the creation relation among contracts, i.e. which create which. This would mean that the final database design for the Contract Creation database would look as illustrated in Figure 17. Since the structure is like a hierarchy of owners creating contracts and possibly contracts creating contracts, it will naturally have a tree-like structure. The top *Ownership* category will contain all *EOA*s as subcategories. In addition, all contracts will be subcategories to their creator. This may be either an EOA or a contract.

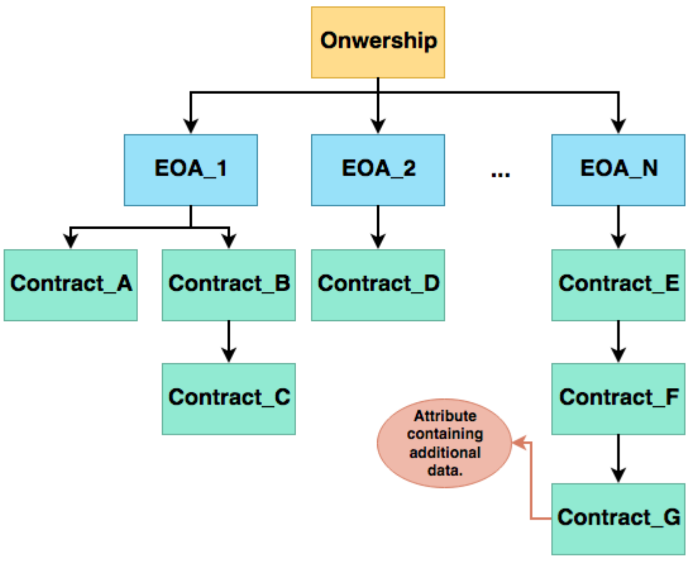


Figure 17. The design of how the contracts are stored grouped with their owners.

Each contract must contain additional information about the contract creation, such as the contract code hash, the nonce, the time stamp of its creation and its value. The red circle in Figure 17 shows how this attribute should be played out. This design is useful because it shows the lengths and depths of contract creations without storing additional duplicates.

3.5.3 Designing a Storage Model that Constructs the Structure of each Contract-to-Contract Invoke

Another goal is to investigate invokes among contracts. Anyone can potentially create a contract that invokes a separate contract, meaning it is possible that the calls among contracts create tree-like structures with cyclic references. Figure 18 is an example of such a structure.

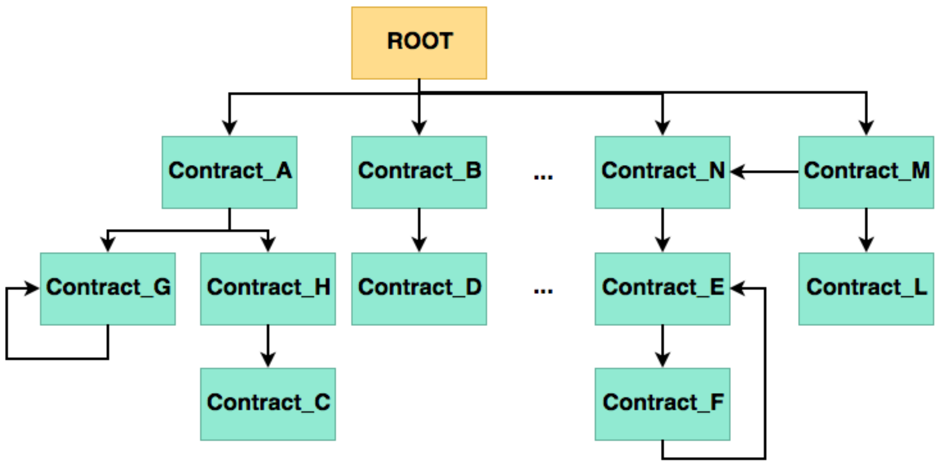


Figure 18. The structure of the invokes tree of contract-to-contract calls.

The contracts do not necessarily share the same owner, even though they are in the same sub-tree. Each sub-tree reflects a single transaction performed, meaning that the top contracts include a call to their sub-contracts in their code. Likewise, the sub-contracts must include a call to the sub-contracts, but the bottom contracts do not necessarily have to be dependent on any additional contracts. This is, however, not the case for *Contract_F*. Even though it is the bottom contract, *Contract_F* makes a call to the contract before it, meaning that both contracts include references to one-another making them co-dependent. *Contract_M* refers to an example of a top contract that makes a call to *Contract_M* and *Contract_L*, both being independent sub-trees.

3.6 Preprocessing Steps

In order to get to an invokes tree of contract-to-contract calls, there is a preprocessing step that requires additional data sorting and storage. As mentioned earlier, one of the main goals is to achieve Ethereum transactions sorted against three overall types; contract creations, contract-to-contract invokes and Ether transfers. In addition, it is also a goal to separate Tokens with regular transfers and contract-to-contract invokes.

3.6.1 Token List

BigQuery has a file section called `token-transfers`, which contains transaction hashes and contract addresses of all tokens on Ethereum. To get a list of all Tokens requires a preprocessing step that runs through all `token-transfers` files and saves each contract address to a separate file.

3.6.2 Contract List

Likewise, the Token list, creating a list of all contracts also requires a preprocessing step that runs through all `transactions` files and saves each contract address, linked to a contract creation, to a separate file. This is a necessary step since each account address, whether it is an EOA or a contract account, are both 20-byte hex addresses that cannot be individualised. To sort EOAs against contract accounts a check is required, which is what this list should provide.

4. Design

The following section explains the overall architecture of the framework along with the overall design and the execution process.

4.1 The Architecture of the Framework

Figure 12 is a pipes and filters architecture of the overall structure of the entire framework. The black pumps on the left are all the transactions files and token-transfer files from the entire Ethereum blockchain. The token-transfer files are used once, to process a list of all token contract addresses. The transactions files are used for multiple purposes. First to create a file containing all contract addresses that exist on the blockchain. Second to filter through the Contract Creator, whose job is to sort contracts concerning their owners and run hash computations against the contracts to get hash strings for each contract. The Contract Creator filter will process all transactions and extract the needed information and run checks against the contract address file. Owners will be grouped with their contracts and saved to a CC Database, after which it is filtered once more to create clustering visualisations of the owner and contract relationship.

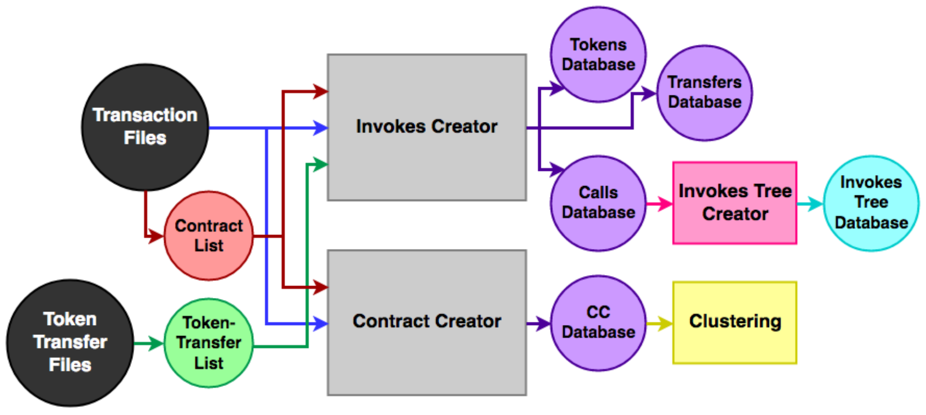


Figure 19. The architecture of the framework.

The third use of the transactions files is to process and sort all transactions that are not contract creations. This filter goes by the name Invokes Creator and additionally uses the list of contracts and the list of token contract addresses to run checks. The preprocessing then sorts all transactions into three databases, Tokens, Transfers and a database called Calls which contains all contract-to-contract transactions.

4.2 Execution Process

As mentioned in Analysis, the amount of data being processed is large. The architecture of Figure 20 will run through the entire data sequentially. However, that would take a lot of time. Next is a demonstration, in the rest of this section, of a parallel execution process that splits the data into N chunks and runs it on N processors, to save time.

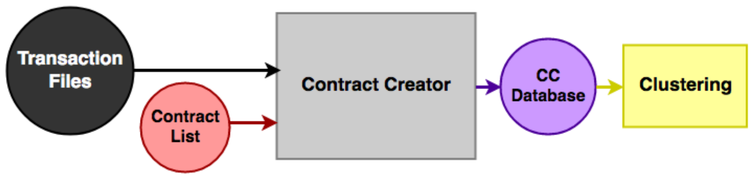


Figure 20. The execution process of collecting contract creations and clustering.

Figure 20 is the execution process of running Contract Creator and clustering. The number of contract creations are 2.387.110 out of 408.137.399 total transactions, meaning that 0.58% are contract creations. This process should, therefore, run on a single processor.

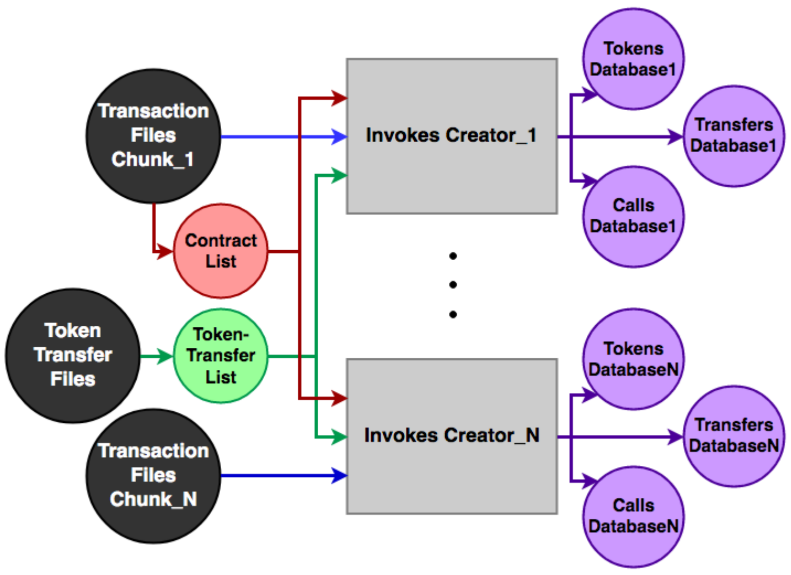


Figure 21. The execution process of extracting contract-to-contract calls, transfers, and tokens.

Figure 21 is the execution process of extracting contract-to-contract calls, transfers and tokens using Invokes Creator. This step processes the majority of the data and therefore requires a lot of time. The end result does not have to be in a single database, so it is an option to divide this process into chunks. The process of Figure 21 divides the data into N chunks and ends up with $3N$ databases. Once this process is finished Invokes Tree Creator then processes each database with contract-to-contract calls and creates a collected tree of all invokes between contracts. Figure 22 illustrates this process.

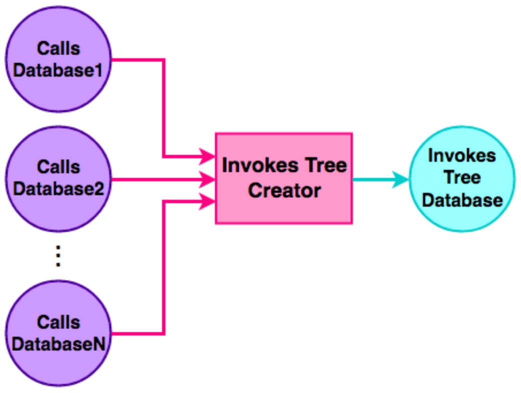


Figure 22. The execution process for creating an invokes-tree using the contract-to-contract calls database.

5. Technical Description of the Program

5.1 Storage

To meet the requirements of the database design, HDF5 (HDFGroup, 2011) offers a hierarchical database structure that performs well with big data. HDF5 is compatible with programming languages such as Java, MATLAB, Scilab, Octave, Mathematica, IDL, Python, R, Fortran and Julia.

The benefits of using HDF5 as opposed to relational databases, like SQL is that HDF5 has a compact format that will conserve file space and process overhead when working with small groups. The group structure of HDF5 can have links that create cyclic references, as demonstrated in Figure 23, which is well suited to create the contracts-to-contracts invokes tree, in addition to the contracts grouped concerning their owners.

This structure could be applied to a relational database, however, it would cause duplicates in the database, thus increasing the size of the storage. Additionally, it would also be more difficult to extract each sub-tree of invokes, as it would require elaborate querying strategies.

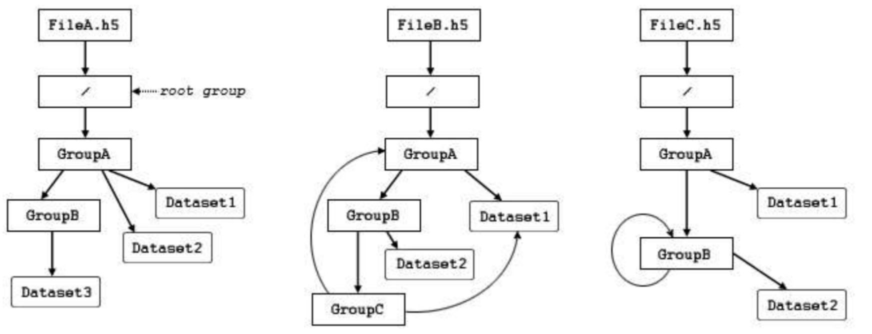


Figure 1. An HDF5 file with a strictly hierarchical group structure

Figure 2. An HDF5 file with a directed graph group structure including a circular reference

Figure 3. An HDF5 file with a directed graph group structure and one group as a member of itself

Figure 23. The HDF5 hierarchical and cyclic grouping options (HDFGroup, 2011).

The disadvantage of using HDF5 is that it can only have one read or write at a time, which means that there can either be one database that is read or written to sequentially or there can be many databases that are read or written to in parallel.

This would not be a problem with relational databases, since they are transactions based, meaning they effectively can have multiple reads and writes at a time.

5.2 Programming Language

5.2.1 Desired Properties

As the goal of this thesis is to construct a framework, it would be beneficial to use a language that can be applied by a high range of people. It should additionally have well functioning data science libraries, work with HDF5 and be able to handle large data sets.

5.2.2 Chosen Programming Language

Among the languages that HDF5 supports. Python is the chosen language for this framework due to its powerful libraries that support both using HDF5 and data science libraries.

One of the disadvantages of Python is that it is an interpreted language and compiled languages are always computationally faster. However, the runtime is dominated by reading and writing to the disk, so the fact that Python is not computationally fast, will probably not be significant enough to avoid using Python's compared to the benefits that Python has. Another disadvantage that Python has is its global interpreter lock, meaning Python will always run on one thread which excludes the option of multi-threading unless external tools are run in parallel threads. Where external tools for instance refers to libraries that do not use Python or network requests, that too does not run Python.

5.3 Applied Tools

The Python library extension of HDF5 is called h5py (Collette & contributors, 2018), where h5py groups are Python dictionaries and h5py datasets are NumPy arrays.

Scikit-Learn (*Scikit-Learn*, n.d.) is a tool for performing Machine Learning in Python built on NumPy, SciPy, and matplotlib. Scikit-Learn has a built-in KMeans clustering along with a set of functionalities to implement the Elbow Method as mentioned in 3.5.1 Clustering

5.4 Available Hardware

The framework is built on access to three servers provided by the University of Copenhagen. The specification of the servers is:

- 16 cores
- 2-way multithreading
- 128 GB DRAM

5.5 Difficulties

One of the difficulties of using Python and HDF5 was getting optimal performance. This was a difficult task due to the combination of Python lacking the option of multithreading and HDF5 only being able to receive one read or write operation at a time. The first solution involved network requests, as explained in Section 7, and for this solution multithreading was applied. However, this was not a stable solution, because there were many failures and time-outs with the network, which is why the second solution excluded all use of the network.

Instead, the best solution for optimal performance was to take advantage of the hardware and split the data on each available server running 5-6 processes each. All the processes were run from a *home* folder which was under NFS, meaning it was not always stable and excessive use of threads was likely to make it crash. For this reason, 5-6 processes were the most stable and yet still fast outcome.

The difficulties involving the final solution was handling the databases. As mentioned previously, HDF5 can only have one read or write at a time, which meant that it was necessary to create 153 databases in total, for sorting data the did not involve Contract Creations. 4.2 Execution Process describes how this execution process was designed.

6. Guide for Running the Framework

The code and clustering results are publicly available on *Github*[7]. The Ethereum data and databases are provided upon request.

6.1 Prerequisites and Requisites

The framework is built on Python 3.7 with Anaconda[8]. Run `init.py` to install additional libraries not included in Anaconda.

6.2 Steps of Execution

There is a Makefile containing commands to run each step. The following is the order of the program execution.

1. Create the files containing all contract addresses and all contract addresses that are tokens, respectively. This step is essential for the framework to run, as it requires multiple checks against the information given in both files.
   ```
   command:  make createcontractlist
   command:  make createtokenlist
   ```
2. Prepare and initialise the databases.
   ```
   command:  make cleanall
   ```
3. Preprocessing step of creating the databases. This part is split into four, where it is suggested to run it on three servers, adding command four below, to any one of three servers.
   ```
   command:  make createinvokes1
   command:  make createinvokes2
   command:  make createinvokes3
   command:  make createcc
   ```
4. Once the databases are finished, one can find the right number of clusters meanwhile initialising and creating Invokes tree.
   ```
   command:  make numclusters
   command:  make treesetup
   command:  make invokestree
   ```
5. Running clustering.
   ```
   command:  make clusters
   ```

[7]https://github.com/smhyatt/ethereum-analytics-framework
[8]https://www.anaconda.com/

7. Experiments

The first solution was a combination of the exported data from BigQuery and applying the Web3 API to verify which addresses were EOAs or contract accounts. Since this solution required a lot of work on the network, it made sense to use multi-threading in Python. Which due to Python's global interpreter lock, would not make sense otherwise.

Below is an illustration of the three threaded pipeline created to speed-up the preprocessing stage of the data. The boxes marked with dark orange represents a method whose job was to sort all the transactions that were not Contract Creations since these were by far the majority of the transactions among the data, this was a step that required a lot of network to process each address in the transaction data.

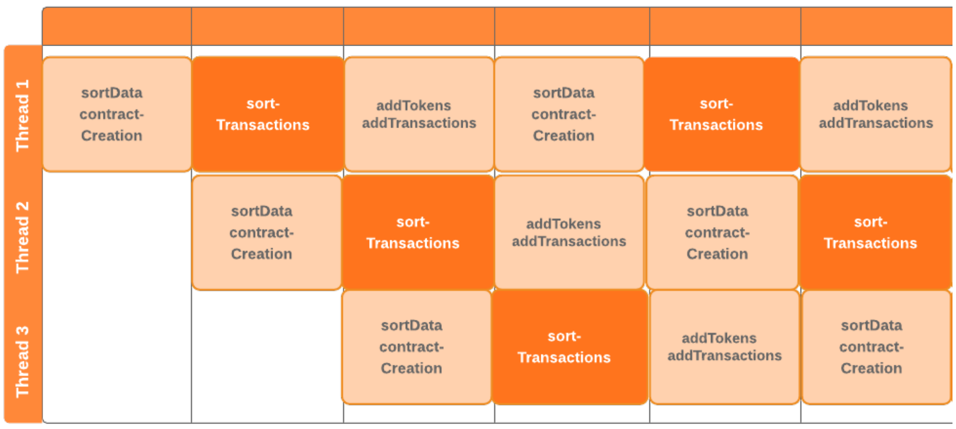


Figure 24. Pipeline of the first solution using the network.

Although the above was a working solution, it had very low performance due to the high use of network connections with the Web3 API. Performing a time experiment on three small files with a total size of 2.42 MB it took 340.825 seconds for the first solution to perform the preprocessing step, meanwhile, the final solution without network takes 11.093 seconds to preprocess the same data. This gave a speed-up of factor 31.

However, it must also be taken into account that the final solution has its own preprocessing step. This includes running through all the transaction files and collecting each created contract address. Between the first solution and the final, there was also an additional speed-up step. To begin with, this file containing all contracts would be loaded into a list in the program. However, changing this list to a dictionary instead managed to increase the performance from 259.207 to 16.782 seconds, giving a speed-up factor of 15.

Dividing the work of the preprocessing step onto three servers, whose hardware is specified in 5.4 Available Hardware, using the final solution, has a total runtime of two and a half hours processing 265 GB of data.

8. Evaluation

8.1 Performance

The collected time of execution for processing and examining 408.137.399 transactions was 9,5 hours, when run on the hardware conditions as specified in 5.4 Available Hardware. The examination part includes the clustering process as well as creating the Invokes tree.

8.2 Findings

8.2.1 Clustering Results

The results in Figures 25, 26 and 27 refer clustering equivalence classes with owners that have the same number of contracts, as described in 3.5.1 Clustering. Figures 25 and 26 are the applications of the Elbow Method, which too is described in 3.5.1 Clustering.

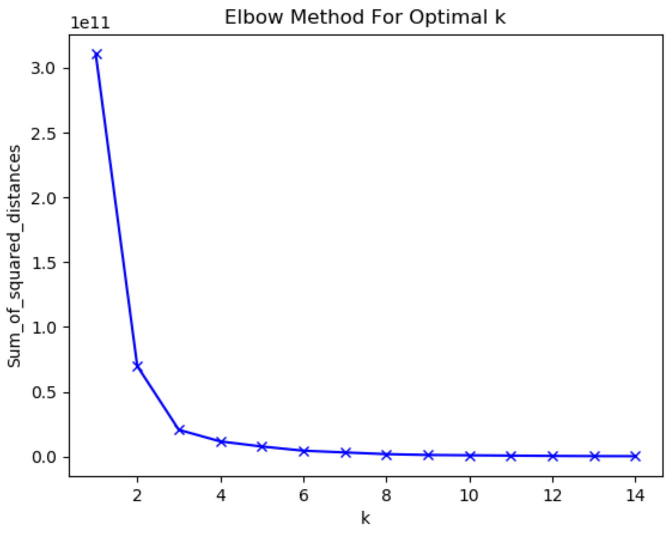


Figure 25. Finding the right number of clusters for the range of contracts that occur per EOA in comparison with the number of times that this amount occurs, using k in the range of 15.

Figure 25 shows the Elbow Method on a k range up to 15, where the point of which the graph bends is around three.

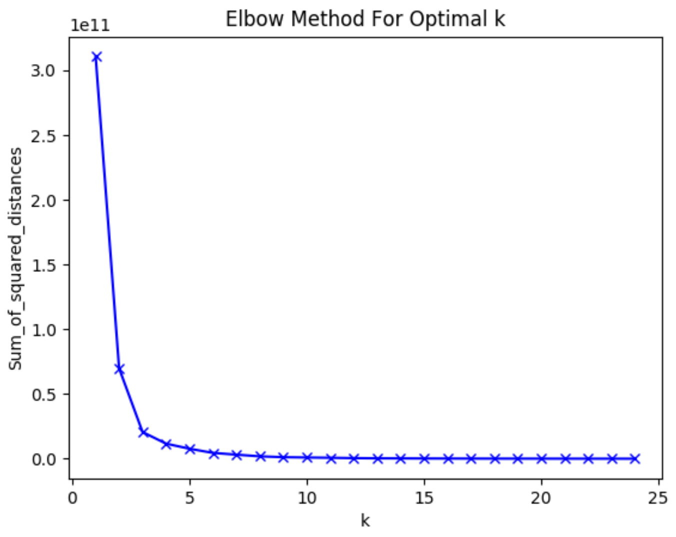


Figure 26. Finding the right number of clusters for the range of contracts that occur per EOA in comparison with the number of times that this amount occurs, using k in the range of 25.

Figure 26 shows the Elbow Method on a k range up to 25, where the point of which the graph bends also is around three. The correct number of clusters must, therefore, be three. Figure 27 shows the clustering result with three clusters. The black spots are centroids[9].

[9]A centroid is the arithmetic mean position of all the points in the figure.

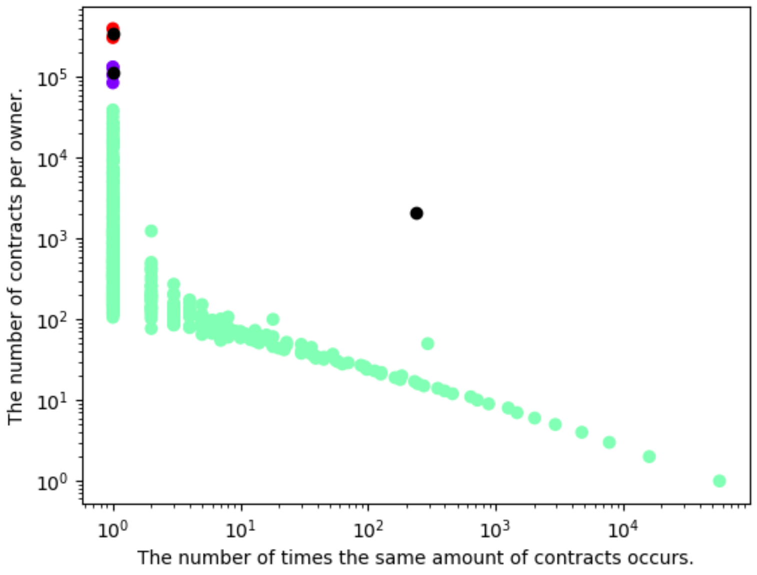


Figure 27. The range of contracts that occur per EOA in comparison with the number of times that this amount occurs, with three clusters.

The clustering result shows that there are few EOAs that have many contracts, because once the number of contracts per EOA decreases, then the amount of occurrences increases. The results also show that the majority resides within the green cluster where the range is approximately between $1-1000$ contracts per EOA.

Figures 28 and 29 are the results of clustering the next point in 3.5.1 Clustering, where the goal is to compare the unique number of contracts per EOA with the number of contracts per EOA. Figure 28 is an additional plot of the Elbow Method on a k range up to 15, where the point of which the graph bends is around three. The Elbow Method has also been applied for k in the range up to 25, however, like in the previous Elbow result, this too showed that the right number of clusters was three.

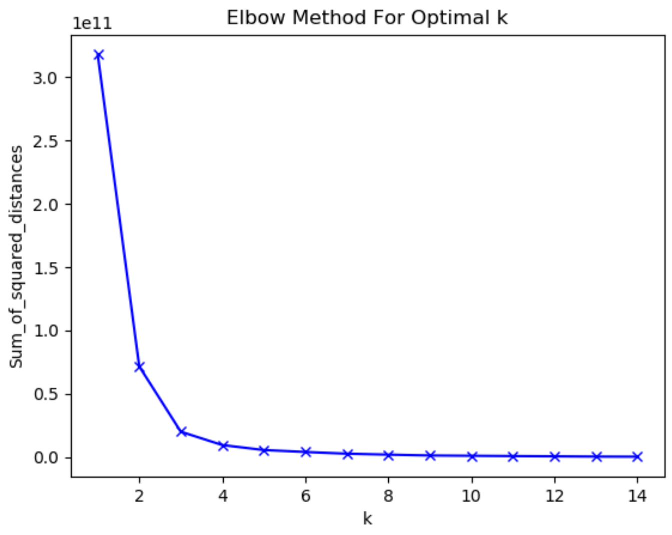


Figure 28. Finding the right number of clusters for the number of contracts per EOA with the number of unique contracts per EOA, using k in the range of 15.

The resulting cluster in Figure 29 shows that there generally are many unique contracts per EOA, which refers to the majority of the purple section.

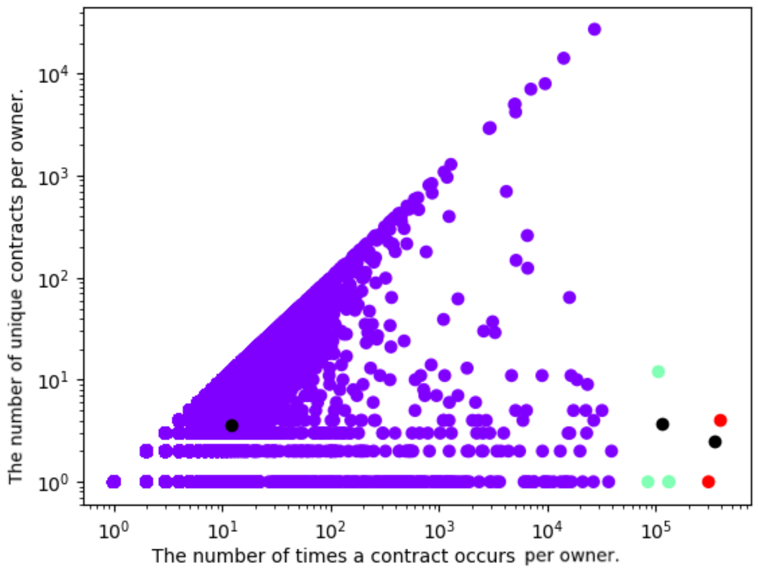


Figure 29. Clustering result of the unique number of contracts per EOA with the number of contracts per EOA, using three clusters.

However, the red group shows that there are occurrences with single owners having more than 100.000 identical contracts. As the number of contracts rises, the unique contracts move towards either many unique or many copies.

Lastly, Figures 30 and 31 refer to comparing the number of times a contract occurs more than once with the number of unique EOAs using each contract. Figure 30 is the Elbow Method applied with k in the range up to 15, where it clearly states that three is the point for the best number of clusters. This too was applied on additional ranges of k, where each result showed that three was the best number.

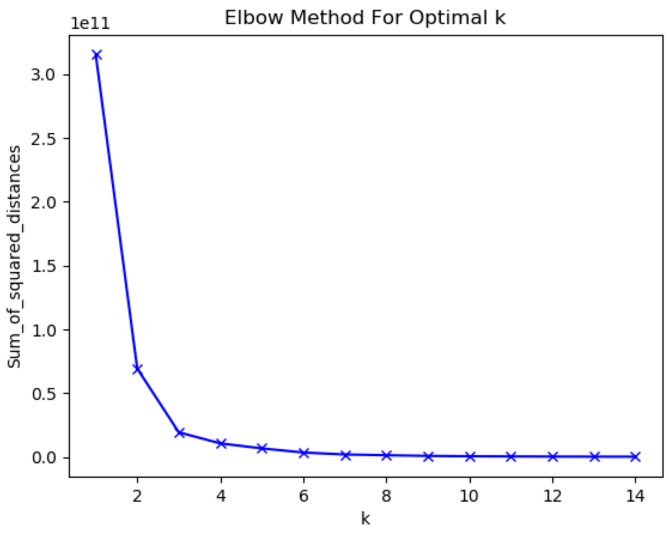


Figure 30. Finding the number of clusters for the number of time a contract occurs more than once with the number of unique EOAs using each contract, using k in the range of 15.

The clustering results of Figure 31 show the occurrences of contract copies among different EOAs. The first occurrence of an EOA is removed, meaning the plot shows all additional EOAs using a contract that has already been in use. The purple area shows that there are some contracts that many different EOAs are using and that there are some contracts that few different EOAs are using. The interesting aspect of this result is that there seems to be a tendency to contracts that many EOAs are copying or either contracts that hardly and EOA is copying.

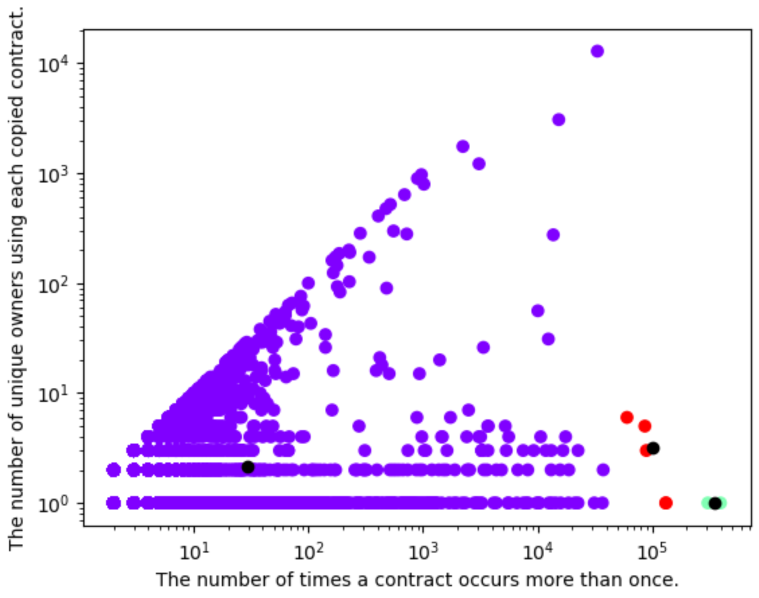
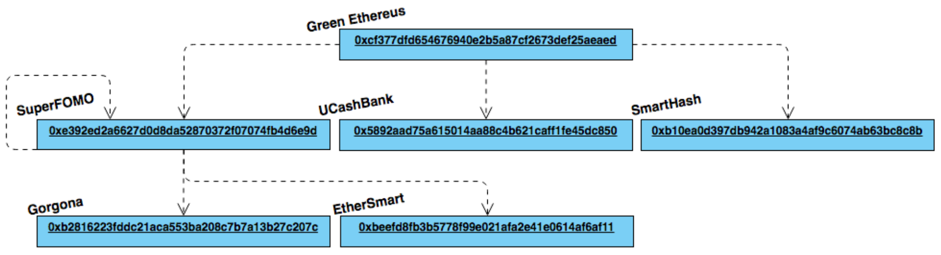


Figure 31. Clustering of the number of time a contract occurs more than once with the number of unique EOAs using each contract, using three clusters.

8.2.2 Invokes tree example

One of the trees that was created by the invokes is the Figure 32 below. In the contract code for address *Green Ethereus* calls to three different contracts are made. *SuperFOMO*, *UCashBank* and *SmartHash*. However, the contract code of address *SuperFOMO* also makes three additional calls. A call to itself, causing a cycle, call to *Gorgona* and a call to *EtherSmart*. The contract for *Green Ethereus* ends up creating this tree-like structure of invokes.



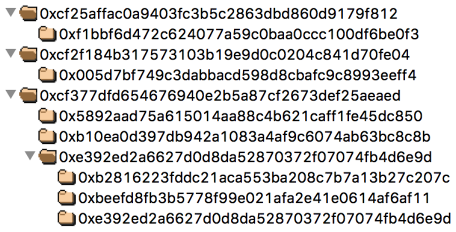
Green Ethereus
0xcf377dfd654676940e2b5a87cf2673def25aeaed
SuperFOMO
0xe392ed2a6627d0d8da52870372f07074fb4d6e9d
UCashBank
0x5892aad75a615014aa88c4b621caff1fe45dc850
SmartHash
0xb10ea0d397db942a1083a4af9c6074ab63bc8c8b
Gorgona
0xb2816223fddc21aca553ba208c7b7a13b27c207c
EtherSmart
0xbeefd8fb3b5778f99e021afa2e41e0614af6af11

Figure 32. Illustration of a transaction tree found in the invokes database, demonstrating a combination of confirmed ponzi schemes.

Looking into the contract addresses of the invokes tree in Figure 32, it became visible that the root contract *Green Ethereus* in fact was a ponzi scheme. Diving further into each of the contracts called by *Green Ethereus*, they too turned out to be separate ponzi schemes.

8.3 Improvements

The Invokes Creator does not create circular reference links between contracts, the current solution writes the address again as a sub-group as shown in Figure 33. Nevertheless, this is a fairly simple extension was not implemented due to lack of time.



```
▼ 0xcf25affac0a9403fc3b5c2863dbd860d9179f812
    0xf1bbf6d472c624077a59c0baa0ccc100df6be0f3
▼ 0xcf2f184b317573103b19e9d0c0204c841d70fe04
    0x005d7bf749c3dabbacd598d8cbafc9c8993eeff4
▼ 0xcf377dfd654676940e2b5a87cf2673def25aeaed
    0x5892aad75a615014aa88c4b621caff1fe45dc850
    0xb10ea0d397db942a1083a4af9c6074ab63bc8c8b
  ▼ 0xe392ed2a6627d0d8da52870372f07074fb4d6e9d
      0xb2816223fddc21aca553ba208c7b7a13b27c207c
      0xbeefd8fb3b5778f99e021afa2e41e0614af6af11
      0xe392ed2a6627d0d8da52870372f07074fb4d6e9d
```

Figure 33. Current invokes tree cyclic reference solution.

9. Conclusion

Overall the goals set for this thesis were met. Inspecting the finalised framework along with the results, the obtained accomplishments were as follows.

– A collection of sorted data, with categories that extend beyond the scope of this thesis. In other words, there is a whole range of sorted token data and Ether transfers that are yet to be explored.
– Clustering results that show significant signs of contract replications along with typical and nontypical equivalence classes of owners that have the same number of contracts.
– Invokes tree that illustrated clear deviations among contract-to-contract invokes, in which a pyramid scheme showed up.

Due to lack of time the CC tree does not have contracts as sub-groups to other contracts. This was not immediately solvable because the sender of a contract creation is always the EOA, even though a contract was created by another contract. It is, however, doable by running checks against each contract account nonce and combining tree patterns from the Invokes tree. The current structure does, however, not have any effect of the clustering, which is why it was not an immediate problem to solve.

9.1 Future Work

- Extending the framework to gather and run the latest blocks from Ethereum. The current solution has the chain data up until block March 18, 2019.
- The results in 8.2.1 Clustering Results demonstrate clear patterns of . Additionally, the clusters all had three groups, it would be interesting to perform clustering on each of these subgroups to understand more of the patterns and structures.
- There exist sites that offer a list of registered scams occurring on Ethereum. Potential future work could include finding and studying the patterns of these scams with the purpose of creating a model that is able to identify future scams.
- The framework has sorted data that can be used to analyse transfers between EOAs, potentially learning where the Ether moves and where it stagnates.
- Extending Invokes tree to include links that show cycles.
- Extending CC tree to have the full design as described in 3.5.2 Designing a Storage Model that Preserves the Structure.
- Creating an extension that runs through each database and 'follows the money'. In other words, analysing the flow of Ether among contract-to-contract invokes and additionally among Ether transfers.
    - Whilst comparing the results with dates of well known attacks.

10. Appendix

10.1 Using Web3 to get the latest block

```
AttributeDict({
'number': 7880743,
'hash':
  HexBytes('0xbfdccc8e28916930e71ded1df813891ec16001b68026a082f7b946342d89c2b6'),
'parentHash':
  HexBytes('0xca4d29dc8ab2efcb7bb88099cfe897fb718cff98b36504791999e2a91929af39'),
'sha3Uncles':
  HexBytes('0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347'),
'logsBloom':
  HexBytes('0x0c04408400c02510044324000290d084400410082c1200428d008108028801089420
            0100120060003004001 2c00a0010125960088ca0a0480220090010b400014016248381
            00003848012008104401025004 4c880b0c4808890021019000001 1607c0408ca026221
            4001202121030e84100002494030256026303912448050840 1000c00d50002420d0004
            0400042822044801218100100180010022020470a006000016004022800102012 08100
            a024048494100080c00200804008000401001004a8830004204020000c020840002800
            4068040c0080008c040700100060010290280808000403208800808601020451020414
            01880cc1026800104042c004'),
'transactionsRoot':
  HexBytes('0xf8633c224835b7e3326ae1549993d64624eabb6c15ba15921ac4521f301783b5'),
'stateRoot':
  HexBytes('0xd63199c45a4b6964dbf48f030203fde262763f7b1ebbbd00d03888e580257096'),
'receiptsRoot':
  HexBytes('0x90845867dd243daac07ec62a78c440c1350a6e397ceaf2a0f1aaee1c45d26cc8'),
'miner': '0x52bc44d5378309EE2abF1539BF71dE1b7d7bE3b5',
'difficulty': 2127015722219317,
'totalDifficulty': 10447291740028362513397,
'extraData': HexBytes('0x50505945206e616e6f706f6f6c2e6f7267'),
'size': 21888,
'gasLimit': 8003877,
'gasUsed': 8003391,
'timestamp': 1559487182,
'transactions':
[HexBytes('0xda970ed3c54cc334b64344a07fe72ad58b6800510c3dbff8b82e01a5dc0e7972'),
  HexBytes('0xf78c192361b140d1155deb6ef715c76944db0f21300087d7612cc93a029c7cfc'),
  HexBytes('0x45bf76d9bc5fecbe031f0352dca195b83eaee1023b92f14b9b463d12c8d05a97'),
  HexBytes('0xd0a3e8290af2f075a3a9438b7e0894292ae97c14ed99cba93152aee2cc6c4faf'),
  HexBytes('0xe0d8ad7044b4ab564f409ef24d893ee7fcb21c81f9284bb88b17bfab8ff87456'),
  HexBytes('0x93aa6538bf29bdb5f337c1f3bed0e6544c1646644f715328ec45ca615d747850'),
  HexBytes('0x66b4138a0e812c724958ebf2fee87a8146ef1c68820600f7931c8ffd0fc44047'),
  HexBytes('0x2ae3c7a18e81b5f67be557ec0406a1a878681117ddd0af89330dc33356d6d35b')
(...)
```

Figure 34. An example of retrieving the latest block on Ethereum, using Web3.

The last part of the output continues with additional transaction hashes.

References

Amani, S., Bégel, M., Bortin, M., & Staples, M. (2018). Towards verifying ethereum smart contract bytecode in isabelle/hol. , 66–77. doi: https://dio.org/10.1145/3167084

Bergel, A., Cassou, D., Ducasse, S., & Laval, J. (2013). *Deep into pharo.* Square Bracket Associates.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., ... Zanella-Beguelin, S. (2016). *Formal verification of smart contracts* (Tech. Rep.). Harvard University and Microsoft Research and Inria. doi: https://dl.acm.org/citation.cfm?id=2993611

Bragagnolo, S., Rocha, H., Denker, M., & Ducasse, S. (2018). Smartinspect: Solidity smart contract inspector. , 9–18. doi: https://dio.org/10.1109/IWBOSE.2018.8327566

Brant, J., Lecerf, J., Goubier, T., Ducasse, S., & Black, A. (2017). *Smacc: a compiler-compiler.* The Pharo Booklet Collection.

Bryant, R. E., & O'Hallaron, D. R. (2011). *Computer systems - a programmer's perspective.* Pearson Education, Inc., 501 Boylston Street, Suite 900, Boston, Massachusetts 02116: Pearson Education Inc.

Buterin, V. (2013). *Ethereum white paper* (Tech. Rep.).

Collette, A., & contributors. (2018). *h5py documentation* (Tech. Rep.).

Delmolino, K., Arnett, M., Kosba, A., Miller, A., & Shi, E. (2015). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.

Dinh, T. T. A., Liu, R., Zhang, M., Chen, G., Ooi, B. C., & Wang, J. (2017). *Untangling blockchain: A data processing view of blockchain systems* (Tech. Rep.). IEEE.

Ethereum homestead. (n.d.). Retrieved from `http://www.ethdocs.org/`

Ethereum market cap. (n.d.). Retrieved from `https://coinmarketcap.com/currencies/ethereum/`

Exporting table data from big query. (n.d.). Retrieved from `https://cloud.google.com/bigquery/docs/exporting-data?fbclid=IwAR0Ke93u61BzekGuBM1efr5GyX6FTGcH5iK6MFSSHYZrvsN3xWP1KmJw-4c`

Fernandes, S., & Bernardino, J. (2015). What is bigquery? doi: https://dl.acm.org/citation.cfm?id=2978309

Grus, J. (2015). *Data science from scratch.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc.

HDFGroup, T. (2011). *Hdf5 user's guide* (Tech. Rep.).

Infura. (n.d.). Retrieved from `https://infura.io/`

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. , 254–269. doi: https://dl.acm.org/citation.cfm?id=2978309

Nakamoto, S. (2009). *Bitcoin: A peer-to-peer electronic cash system* (Tech. Rep.). doi: www.bitcoin.org

Nicola Atzei, M. B., & Cimoli, T. (2017). *A survey of attacks on ethereum smart contracts* (Tech. Rep.). Università degli Studi Cagliari, Italy. doi: https://eprint.iacr.org/2016/1007.pdf

Peck, M. E. (2017). Blockchains: How they work and why they'll change the world.
doi: 10.1109/MSPEC.2017.8048836

Prasad, R. V., Dantu, R., Paul, A., Mears, P., & Morozov, K. (2018). A decentralized marketplace application on the ethereum blockchain. , 90–97. doi: 10.1109/CIC.2018.00023

Rocha, H., Ducasse, S., Denker, M., & Lecerf, J. (2017). Solidity parsing using smacc: Challenges and irregularities. , 1–9. doi: 10.1145/3139903.3139906

Scikit-learn. (n.d.). Retrieved from `https://scikit-learn.org/stable/index.html`

Solidity. (n.d.). Retrieved from `https://solidity.readthedocs.io/en/v0.4.25/`

Szabo, N. (1997). *Formalizing and securing relationships on public networks* (Tech. Rep.).

Web3. (n.d.). Retrieved from `https://web3j.readthedocs.io/en/latest/`

Wood, D. G. (2014). *Ethereum: A secure decentralised generalised transaction ledger* (Tech. Rep.). doi: https://gavwood.com/paper.pdf

Wöhrer, M., & Zdun, U. (2018). Smart contracts: Security patterns in the ethereum ecosystem and solidity.
doi: 10.1109/IWBOSE.2018.8327565