DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Conceptual Model for Ethereum Blockchain Analytics

Alexander Hefele

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Conceptual Model for Ethereum Blockchain Analytics

Ein konzeptuelles Modell zur Analyse der Ethereum Blockchain

Author:	Alexander Hefele
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	February 14, 2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, February 14, 2019 Alexander Hefele

Acknowledgments

First of all, I would like to thank my thesis advisor Ulrich Gallersdörfer for his continued support throughout the entire time of my thesis. I am especially grateful for the time he took for our weekly meetings that often went beyond just discussing the progress of my work. Furthermore, I explicitly thank him for the countless times he proofread parts of my thesis and gave valuable feedback.

Likewise, I want to thank Prof. Dr. Florian Matthes for his time and ideas that went into shaping the topic of this thesis. I am very thankful for the opportunity to write this thesis at his chair for Software Engineering for Business Information Systems (SEBIS).

Undoubtedly, this thesis would not have been possible without the support of my parents. I thank them for their financial support and their encouragement throughout my studies.

Kind acknowledgments go to my roommates Larissa, Yara, and Michael for their patience and support, especially during the last weeks of my thesis.

Yet, my last thank you is to Simone who always managed to distract me after a stressful day with her pursuit of sports activities. Without her motivation, which had such a large impact on me, this accomplishment would not have been possible. Thank you.

Abstract

In this thesis, we develop a comprehensive formal model representing the Ethereum platform in the form of UML class diagrams. Splitting the system into the four parts “Source”, “EVM”, “Storage”, and “Ledger” helps us to bring a clear structure into this complex environment. These four parts aim to give a deep understanding of the contract-programming language Solidity, the underlying Ethereum Virtual Machine, how each node in the network stores account and state information, and the contents of the blockchain itself.

Afterwards, we apply our knowledge about the system and explore what data can be extracted from the Ethereum platform, and how this can be done efficiently. In the relational database that we build up, we store bytecodes and additional information of all user- and contract-created smart contracts from the first 6,900,000 blocks.

With this data, we perform different analyses to gain more insights into the system. The researched anomalies include front-running, self-destructing constructors, and transactions to accounts that only become contracts after the transaction has been executed. Additionally, we cluster smart contracts based on different criteria, like who created them and whether they implement ERC token standards. Consulting metadata information, like references of hard-coded addresses in the bytecode of contracts, the usage of certain function signature hashes, and the balances of contracts that a contract created, further refines our system understanding.

The main contribution of this work is the estimation of compiler and Solidity library versions of arbitrary smart contracts. With two heuristics based on the contract creation date and the bytecode header, we set a range of minimum and maximum compiler versions for every contract code. We discover usage of the most popular Solidity library “SafeMath” by compiling every version of the library with every compatible compiler version, extracting its internal functions, and comparing the resulting bytecodes with all contract codes deployed on the blockchain. That also helps us improve the compiler version estimation.

We evaluate our version estimations with verified contracts from the block explorer website Etherscan. For our compiler version estimation, the range we set is correct for 99% of the evaluated contract codes. The median size of the estimated compiler version range is 3. For SafeMath usage detection, we have a success rate of 82% with a median distance of 4. Despite considering 31 SafeMath versions, the highest library distance our approach sets for a contract code is only 14.

Contents

1 Introduction

1.1 Motivation

The rate at which a society makes progress is often underestimated. When trying to predict the future, usually a reference point in the past is taken and then linearly extrapolated. That intuitive approach severely fails to determine a realistic view of the future. In his essay, *"The law of accelerating returns"* [1] futurist Ray Kurzweil describes a more accurate way to look at the future, the "historical exponential view". He points out that for many fields technological change is exponential and even the rate of growth is itself growing exponentially. It is important to note that this double exponential growth has been there from the very beginning of technology. When a specific paradigm providing exponential growth is at its maximum potential, a paradigm shift happens, which allows the exponential growth to continue. This statement can be applied to many different technological and biological developments, including the exchange of goods.

Since early beginnings humans have traded goods one for another. That method is very basic and its lack of versatility sparked the invention of money. Using widely accepted currencies meant a big paradigm shift that did not reach its full potential until thousands of years later. Only with the ascent of banks, a new payment method came into existence. The paradigm shift is cashless payment allowing people to pay practically anywhere in the world without having to carry physical money. However, that paradigm once more started to exhaust its potential, which, this time, is about trust. Banks are central authorities that must be trusted unconditionally when using a cashless payment system. Until about ten years ago, if one did not want to trust a third-party intermediary, one could not use electronic payment and would have to fall back to paper money. Then, in 2008 another paradigm shift happened. With the invention of Bitcoin [2] cashless, trustless payments became possible. At first, the number of users and transactions in Bitcoin grew exponentially, just as the law of accelerating returns predicts. However, after a few years that growth was not sustainable anymore and Bitcoin quickly reached a limit, especially in terms of the amount of transactions that it can process per day.

The next paradigm shift is towards not just trustless payments, but more sophisticated contracts. In 2014, with Ethereum the concept of smart contracts was introduced [3]. These programs running on a decentralized "world computer" allow for complex payment schemes that only require the contractual partners to trust the code that they themselves can verify. Today, Ethereum is the second largest cryptocurrency by market capitalization at currently about 12 billion dollars[1]. The Ethereum network currently processes twice as

[1] `https://coinmarketcap.com/coins/`

many transactions per day as Bitcoin[2].

Although Ethereum is such an established cryptocurrency by now, we still lack a deep understanding of what exactly is happening in the network. Even for such a fundamental metric like the total number of smart contracts, it is hard to find a reliable, precise, and current source. More specific questions, like what smart contracts are most often used for, are still unanswered. To enable a better understanding of the ecosystem we develop a conceptual model of the Ethereum platform, which aims to provide a complete, in-depth overview of the system. Then, we leverage this model to analyze a variety of different aspects of the Ethereum platform. Throughout this thesis, we explore the potential of this most recent paradigm in the field of payment methods.

1.2 Research Questions

We identify the following five research questions (RQ) to cover during this thesis:

RQ1 – How are the different parts of the Ethereum system correlated with each other?

We are looking for a way to bring structure into the complex Ethereum platform. The system can be divided into four distinct parts: “Source”, “EVM”, “Storage”, and “Ledger”. Each part is self-contained and exposes relations to the other parts. When developing our model in chapter 4, we create one UML class diagram for each of the four parts and highlight how these correlate with each other.

RQ2 – What data can be extracted from the blockchain for analysis and how can this be done efficiently?

A crucial requirement for our analysis of the Ethereum blockchain is data acquisition. We investigate what data we can collect from the blockchain and from other relevant sources. That data should then be stored in a permanent way to make analyses on the data fast and reproducible. An important aspect here is efficiency of the data storage, both in terms of accumulating and requesting data.

RQ3 – What does metadata tell us about the network?

Metadata information gives more insight into smart contracts and the general usage of the Ethereum system. We examine several pieces of metadata information, like what addresses are referenced most often from smart contract bytecode. Most importantly, however, our goal is to give an estimation of the compiler version that was used to compile a smart contract, solely by looking at the bytecode. Further, we research whether libraries were used in the contract code.

RQ4 – What are different areas of application of the Ethereum blockchain?

In order to get a complete picture of the Ethereum platform, we research real-world use

[2]`https://coinmetrics.io/charts/#assets=eth,btc_left=txCount`

cases. Our focus is on implementations of token standards and their development over time. Additionally, we inspect smart contracts whose code is replicated on the blockchain most often and examine their purpose in detail.

RQ5 – Which anomalies can be observed in the network?
Besides looking at the normal behavior of the system, we also investigate unusual behavior. Sometimes, relevant events are not the ones that happen regularly, but only occasionally. We detect such behavior and analyze why it happens. One example of an anomaly is front-running.

1.3 Approach

For our research, we follow the Design Science Research (DSR) methodology. This approach is described by Hevner et al. [4] with seven guidelines that need to be followed. First of all, our thesis produces an artifact in form of a model of the Ethereum system. In section 1.1, we showed why this topic is important and relevant. We demonstrate the utility of our artifact by applying the model to the Ethereum blockchain and creating analyses on real-world data, which we later on evaluate based on a data set of verified contracts. All of our contributions to the design artifact are verifiable as we precisely record our rigorous methods for creating the model, our analyses, and the evaluation. This, as well as our search process in form of an extensive literature review is communicated in the present thesis.

1.4 Outline

The rest of this thesis is structured as follows. In chapter 2, we define all necessary fundamental concepts related to blockchain and Ethereum. Chapter 3 gives an extensive overview of related literature. Next, we present our conceptual model and describe every one of the four identified parts in detail. In chapter 5, we describe our data acquisition strategy before we analyze the gathered data in the following chapter. That encompasses general statistics, detection of anomalies, research about areas of application of the Ethereum platform, as well as a metadata analysis, including compiler and library version estimation. We evaluate our main results in chapter 7 and conclude the thesis with a summary and an outlook in chapter 8.

2 Foundations

This chapter gives an overview of some fundamental terms and concepts forming the basis of our conceptual model of the Ethereum platform that we develop later on. Not only do we define the underlying principles about blockchain in general, we also have a look at other distributed ledger technologies, implementations of the Ethereum protocol, and talk about conceptual modeling in general.

2.1 Cryptographic Basics

Before discussing what a blockchain actually is, we first need to present some basic cryptographic building blocks upon which blockchains heavily rely on.

2.1.1 Cryptographic Hash Functions

A hash function is a non-injective mapping from inputs of arbitrary size to outputs of fixed size: $h : X^m \rightarrow X^n$. In general, the image of the hash function is much smaller than the preimage: $n < m$ [5]. Because of that, collisions can occur, meaning that two different inputs are mapped to the same output value. For most use cases, collisions are usually bad and a *cryptographic* hash function is a hash function that fulfills three additional requirements to make collisions occur less often:

- **First pre-image resistance** – Given Y in the image of h, it is computationally infeasible to determine an X, such that $h(X) = Y$.
- **Second pre-image resistance** – Given X and $h(X)$, it is computationally infeasible to determine a $Y \neq X$, such that $h(Y) = h(X)$.
- **Collision resistance** – It must be computationally infeasible to find two inputs X and Y with $X \neq Y$, such that $h(X) = h(Y)$.

Some well-known cryptographic hash functions are MD5, SHA1, and its successors SHA2 and SHA3. While the first two are outdated by now and collisions have been found for them, the latter two are widely used today for all kinds of purposes. For example, comparing the hash of a downloaded file ensures its integrity. Also, in order not to store user passwords in clear text, only the hash of the password should be stored, which is always compared against when a user logs in to the service. And finally, the proof-of-work algorithm used by many cryptocurrencies heavily relies on finding a certain hash value (see section 2.2.1).

2.1.2 Public-key Cryptography

Another use case of cryptographic hash functions are digital signatures. These work on the basis of asymmetric cryptography where there is a public and a private key. The public key can be shared with anyone and can be used to encrypt messages, while the private key must be kept secret by the key owner because it is the only way to decrypt messages encrypted with the public key. The private key can also be used to sign messages. These signatures can then be verified using the public key. Most digital signature algorithms require that the message, which should be signed, is hashed beforehand. Then the signature is calculated only over the fixed-size hash, which is much more efficient than signing the entire message.

Signatures are important for verifying the authenticity of a message in a network. Furthermore, the integrity can be assured and the sender cannot claim that they did not send the message (so-called “non-repudiation”). One very widely spread example for a public-key cryptography scheme that can also be used for signing is RSA. But there are also dedicated signing algorithms, like DSA or ECDSA, which have the advantage of much shorter signatures than RSA at the same level of security.

2.1.3 Merkle Patricia Trees

A Merkle Patricia tree (or short “trie”, originating from the word “retrieval”) is a search data structure for storing binary data of arbitrary length. Its main purpose is to provide a single value that represents the entire set of the stored key-value pairs. The data structure is described formally in the Ethereum yellow paper in appendix D [6] and we only give a non-formal description here.

Each data element that should be stored is split into small chunks, so-called nibbles, which are four bits long. That means there are 16 different values that these nibbles can take. Now the first nibble of all data elements that should be stored is compared and there are two possibilities: Either the first nibble of all data elements is the same, or there is at least one difference. If they are the same, the next nibble is compared until a difference is found. These same nibbles are then combined into a so-called extension node. If there is a difference, a branch node is inserted that links to new nodes each representing the next distinct nibble. In total, a branch node can have at most 16 child nodes. Every data element is then put in the according child branch and the algorithm is performed again for every branch. Lastly, if for a certain branch there is only one data element left, a leaf node is inserted, which contains any remaining nibbles of that key and the value associated with it.

Every node in the trie has a deterministic cryptographic hash value, which is used to link nodes together. In order to represent the entire trie, the hash of the root node can be taken because the root node depends on all other nodes in the trie and any modification in the trie changes the hash of the root node.

2.1.4 Bloom Filter

A bloom filter is a fixed-size bit-array that can be used to check for set membership. Its usage is often more efficient than comparing all elements one-by-one, but the only assertion that can be made with absolute certainty is whether an element is *not* part of the set.

All of the bits in the bit-array are initially set to 0 at the beginning. Every time an element is added to the set, multiple hash values are calculated for that element. These values specify in some deterministic way, which bits to set to 1 in the bloom filter. The amount of bits to set to 1 is always the same for every element. If a bit has already been set to 1 beforehand, it remains unchanged.

To check if some element is in the set, one computes the same hash values for that element. If at least one of these values in the bloom filter is 0, the element is definitely not part of the set. On the other hand, if all bits of that element are 1 in the bloom filter, the element is in the set with high probability, but it is not completely certain since there might be collisions.

2.2 Blockchain

A blockchain is a linked list, whose elements (called blocks) are cryptographically dependent on their direct ancestors. It can be used to maintain a decentralized digital ledger that is agreed upon by all peers in a network and cannot be changed retroactively.

Each block typically consists of a block header and a transaction list. The block header contains the hash of the previous block, a timestamp, and other meta information, depending on the application. In the transaction list, all the transactions between two or more parties (called accounts) are stored. A transaction contains information that updates the ledger, i.e. information about a certain value that is transferred between accounts. A blockchain can be seen as a state machine and a transaction transitions the system from one valid state to another.

2.2.1 Consensus Mechanisms

In order to decide who is allowed to append the next block to the chain, a random peer in the network has to be picked. The consensus mechanism is responsible for this task. Nodes that are trying to create new blocks are called miners and their incentive to mine new blocks is usually a block reward, which they get to transfer to a beneficiary of their choosing when they are the first to mine a new block. There are many different consensus algorithms, of which the proof-of-work algorithm is the most widely spread one.

- **Proof-of-work** – To generate a valid block, a certain computationally complex task has to be solved. Often, this includes finding a value such that a hash over the block or parts of it (including that value) fulfills a specific requirement, e.g. that it is lower than some boundary. Verifying that a given solution is correct can be done in a very short time, but because of the first pre-image resistance property of the cryptographic hash function, finding a valid solution can only be done by trial-and-error. More computing power gives a node a higher chance of finding the solution, mining the block, and ultimately obtaining the block reward. In order to keep block times constant, the difficulty of the task has to be constantly adjusted, in relation to the total computing power in the network.

- **Proof-of-stake** – The idea of the proof-of-stake algorithm is that miners with a high balance or who are part of the network for a long time have a high incentive to protect the cryptocurrency from malicious entities. Therefore, everyone in the network gets to vote on which block to append to the blockchain next, but miners with a higher "stake" have higher influence.

2.2.2 Bitcoin

The cryptocurrency Bitcoin was first presented in a whitepaper in 2008 by an unknown person or group under the pseudonym Satoshi Nakamoto [2]. It heavily relies on a blockchain, which stores all transactions in the network. Shortly after the initial publication, the reference implementation of the Bitcoin client software was released and blocks were starting to get mined. In the beginning, there were only few nodes in the network, and Nakamoto mined about one million bitcoins himself before disappearing [7].

Properties

In the whitepaper, Nakamoto demanded three basic properties that a new digital currency must have [2]:

- It must be based on mathematical proof instead of trust in order to avoid the involvement of a trusted third party.
- To protect sellers, it must not be possible to reverse transactions.
- All transaction must be publicly visible so that double spending can be detected by everyone.

In Bitcoin, a transaction can consist of multiple inputs and outputs. For every transaction, the sender has to specify exactly which portion of the total input value should go to which outputs. However, either all of the balance of the input accounts has to be transferred to the output accounts or none of it. If someone only wants to send a part of their bitcoins to someone else, they have to send a transaction that transfers that part to the other party and simultaneously transfers the remaining coins back to themselves again. This type of system is called a transaction-based ledger and allows nodes having to keep track of only these so-called unspent transaction outputs (UTXOs).

Nakamoto set a fixed block size of 1 MB in his original bitcoin client implementation for unknown reasons. During times of high congestion in the network, not all pending transactions can be included in a block, temporarily resulting in higher transaction fees.

Another value Nakamoto had to define when implementing the first client was the block time. If set too low, the blockchain splits more easily, which happens when two or more miners manage to mine a new block at the same time. Afterwards, each miner has to decide which chain to follow and when one chain has a higher block height than the other, the shorter chain is discarded. Too high block times are an annoyance for sellers because they have to wait longer until they are sure that the transaction is in the correct chain and not in a fork. In the end, Nakamoto settled for a very conservative block time of ten minutes. Decker and Wattenhofer measured the mean block propagation time of the Bitcoin network with 12.6 seconds. After 40 seconds, 95% of nodes have received the block [8]. Therefore, Bitcoin's ten minutes block time could have been easily set much lower without problems regarding block propagation.

Many of the concepts used by Bitcoin are not new, but the way Bitcoin combines them is novel. Being the first cryptocurrency, Bitcoin has on the one hand presented features that are crucial for most cryptocurrencies today, like the concept of a blockchain for keeping track of a distributed ledger. On the other hand it has shown what design flaws to avoid, like too high block times and a too low transaction throughput.

Altcoins

After the release of Bitcoin, many alternatives – so-called altcoins – emerged that tried to fix flaws in the original protocol, or extend it with new features. Altcoins are either created as entirely new currencies (like Monero) or developed as hard forks of Bitcoin (like Bitcoin Cash). Litecoin was released in 2011 and focused mainly on shorter block times, set an increased total number of coins that can be generated, and used a different hashing algorithm. The Bitcoin Cash hard fork increased the block size from 1 MB to 8 MB and later to 32 MB. Another concern with the original Bitcoin implementation is privacy because every transaction on the blockchain is public with sender, receiver, and amount transferred. Monero uses an obfuscated blockchain based on the CryptoNote protocol [9] to disguise these three fields and make transactions untraceable.

2.2.3 The Second Generation Blockchain

While many altcoins improved some specific property of the original Bitcoin protocol, it took until 2014 for a fundamentally different approach to come into existence. Vitalik Buterin proposed a "decentralized application platform" in his whitepaper and called it Ethereum [3]. His cryptocurrency system not only lowered block times and increased the amount of transactions that a block can hold, but introduced the concept of a decentralized computer. This type of system is sometimes referred to as a second generation blockchain [10].

Properties

Several implications arise from the introduction of a decentralized computer. The first one is that every active participant in the network must run all instructions of all programs that need to be run on the computer. Also, everyone needs to have all of the computer's storage stored locally. These implications put a lot of weight on each individual node, but are necessary to have all operations on this "world computer" fully decentralized.

A transaction in this system is a call to a program on the world computer. In order to decide which transactions to run, the blockchain consensus algorithm is used. Programs that run on the blockchain are called smart contracts. Every transaction can call functions, send money, store and retrieve data, and invoke other calls. Due to the tamper-proof nature of the blockchain, once deployed, a smart contract cannot be changed retroactively.

This is only a general description of decentralized computers and smart contracts. We give a more in-depth description of the concrete implementation of these concepts in Ethereum in chapter 4.

2.2.4 Block Explorers

Running a full node in a blockchain network requires a lot of resources, especially in a network of a world computer. For Ethereum, a fast network connection and an SSD is required because classical hard drives cannot keep up with the amount of read and write operations the node needs to make. When one simply wants to look at a specific block, it might not be feasible to set up an entire node for that. Instead, there are online services that allow users to inspect blocks in the web browser. A very popular block explorer for Ethereum is Etherscan[1].

The advantage of block explorers is that they provide fast and easy access to the entire history of all blockchain data. The user does not need costly hardware and can see a user-friendly visual representation of the blockchain with blocks, transactions, accounts, and more. Often, block explorers even provide additional information that a full node cannot offer just like that. For example, every account page on Etherscan contains a list of all transactions that this account sent and received. It is not possible to obtain this information with a standard Ethereum client easily as it requires replaying all transactions in the blockchain.

Using block explorers comes with some disadvantages as well. As all data is retrieved from one website, the user heavily relies on information from one centralized instance. That instance must be trusted when using the service and could potentially give out information that has been tampered with – either deliberately or because the service was compromised by a malicious party. Users of block explorers usually do not run the code to verify its correctness as they would when running a full node. But the entire reason to use a blockchain in the first place was to avoid the involvement of a trusted third party.

In conclusion, block explorers are a convenient way to look at small amounts of data on the blockchain quickly, but for any advanced analyses, data of a full node should be consulted.

[1]`https://etherscan.io`

2.3 Other Distributed Ledger Technologies

Besides using a blockchain, there are other ways to achieve consensus in a distributed system. In general, a distributed ledger is “a type of database that is spread across multiple sites, countries, or institutions, and is typically public” [11]. Participants in the network have to agree upon every record that should be added to that continuous list of entries. We have a look at two of these distributed ledger technologies in this section.

2.3.1 Directed Acyclic Graph

A directed acyclic graph (DAG) is a data structure consisting of vertices and directed edges with the constraint that there are no loops (or cycles) in the graph. Serguei Popov proposed a DAG as an alternative to a blockchain for the cryptocurrency IOTA [12].

In the DAG used by IOTA, which is called *tangle*, each vertex represents one transaction in the network and an edge from vertex A to vertex B indicates that the issuer of transaction A has approved the prior transaction B. Every new transaction must approve and validate two previous transactions and solve a cryptographic puzzle, similar to the one used by Bitcoin, to prevent spam. There are multiple categories of puzzles and if more work has been put into it, a higher weight is assigned to the vertex in the network.

All units of the IOTA cryptocurrency have been created at the genesis transaction, which all other transactions directly or indirectly approve. That means that no more units can be created in the future and there is no mining in the network. Despite that, there are two incentives for nodes to share incoming transactions with the network actively. First, a node usually wants to share transactions that approve one of their own transactions. Secondly, nodes in the network drop inactive nodes that do not propagate new transactions.

Because of the asynchronous nature of the tangle, conflicts can occur. If there are conflicting transactions, the nodes in the network need to decide which transactions become orphaned meaning that they are not indirectly approved by new transactions anymore. The exact algorithm that nodes use to resolve conflicts is out of the scope of this thesis and is covered in [12].

2.3.2 Hashgraph

Another notable distributed ledger technology is the hashgraph [13]. It does not require proof-of-work, is asynchronous, proven to lead to consensus, and has very high throughput. The Hedera Hashgraph Platform[2] is based on this technology.

The foundation of the hashgraph is a gossip protocol where every node in the network picks another node at random and tells that node everything they know about the network. That information is used to build up the hashgraph where each member of the network is represented by a column of vertices. Every vertex corresponds to an event and the higher up a vertex is in the column, the more recent the event is. When one member A receives new gossip from another peer B, this is represented by an edge from the highest vertex in

[2]`https://www.hedera.com/`

B's column to a new, higher vertex in A's column. So essentially, all nodes gossip about who gossiped to whom in what order and that information is stored in the hashgraph.

An event vertex can contain an additional payload, which are transactions that the corresponding member of the network wants to issue. Consensus is achieved because everyone in the network has the same hashgraph. It may only differ for very recent events, but eventually everyone has heard gossip about all events of a certain age. Therefore, with any deterministic function, every node can calculate the same total order of the events and double-spending is prevented [13].

2.4 Ethereum Clients

An Ethereum Client implements the Ethereum protocol. There are several implementations in many different programming languages. The two most widely used clients are Geth and Parity.

2.4.1 Geth

Geth[3] is short for go-ethereum and is developed by the Ethereum foundation. As the name implies, it is written in Go. With the open-source command-line tool, a full Ethereum node can be run. Starting the client is done with the command `geth console`. Afterwards, it is possible to mine Ether, send transactions, create contracts, and investigate block history. Especially the latter is interesting for us since our analysis does not require an active participation in the network. Instead, we passively analyze existing transactions and smart contracts. Geth provides an interface to query specific values from the blockchain (see section 2.4.3).

2.4.2 Parity

The Parity Ethereum Client[4] has very similar functionality as Geth, but it is written in Rust. This relatively new systems programming language aims to achieve high performance and security at the same time. Dr. Gavin Wood, the author of the Ethereum yellow paper, is co-founder of Parity Technologies, which develops the client.

Rouhani and Deters compared the performance of Geth and Parity on a private Ethereum blockchain and observed that Parity processes transactions about 90% faster than Geth [14]. Also, Parity has some advanced features Geth is lacking. Fröwis and Böhme point out that Parity implements a tracing mode, which can be used to tell whether a transaction indirectly created another contract [15]. That is useful to find contracts that were not created by users, but other contracts. However, we show a way to find these contract-created contracts without the use of the tracing mode later on.

[3] `https://ethereum.github.io/go-ethereum/`
[4] `https://www.parity.io/`

In the end, the decision which client to use also depends on personal preferences and external circumstances. For our analysis, we use the existing Geth client setup provided to us. It is a fast-synced node with 32,768MB cache.

2.4.3 JSON RPC and IPC

Both Geth and Parity implement the JSON RPC API [16], which can be used to interact with the blockchain and network from the node. RPC is short for *remote procedure call* and allows obtaining and sending information via any transport method, like *inter-process communication* (IPC) calls, HTTP, and others.

There are five categories of calls, which are identified by their starting letters:

- **web3** – In this category there are two calls: one to get the client version and one to calculate the Keccak-256 hash of provided data.
- **net** – Provides information about the network, like which network the client is listening on (mainnet or one of Ethereum's test networks).
- **eth** – This is by far the biggest category and the most relevant for us. It encompasses all calls that query information about the blockchain, like the current block number and the hashrate if the node is mining. Moreover, information about certain blocks (given the number), transactions (given the transaction hash), smart contracts, and accounts (given the address) can be obtained through calls. For example, the `eth.getCode(address)` call returns the smart contract code that is deployed at the provided address. Furthermore, this category provides calls to compile programs, estimate gas consumption of transactions, and much more.
- **db** – This category has calls to directly retrieve and store strings and binary data in the node's local database. It is mainly intended for debugging purposes.
- **ssh** – These are calls to send and retrieve messages through the ssh whisper protocol. They are of no relevance for our work.

2.5 Conceptual Modeling

According to San José State University professor Jon Pearce, "a conceptual model captures the important concepts and relationships in some domain" [17]. In his lecture about Object Oriented Analysis, he demonstrates how Unified Modeling Language (UML) class diagrams can be used to create a conceptual model. Most fundamentally, every concept should be modeled as a class and relationships are associations between classes. Multiple classes can be grouped together into a package if they are related. We use this approach to design our conceptual model of the Ethereum system in chapter 4.

Evermann evaluated the use of UML class diagrams for conceptual modeling in a study [18]. The main question of his research is whether UML can be used for conceptual modeling and how it should be used. Eventually, he derives a set of rules that, if followed, make UML diagrams very well suitable for conceptual models. Most importantly, Evermann states rules that clearly distinguish objects and attributes: an object should be used to model a substantial entity and attributes should only represent properties of these entities. These suggestions coincide well with Pearce's approach. Furthermore, according to Evermann, classes cannot be solely abstract, but must possess at least one instance.

3 Related Work

In this chapter, we present work related to our research. We identified three broad research areas that are relevant for our analysis.

3.1 Ethereum

On the Internet, there are several resources that try to illustrate how exactly the Ethereum platform works. One popular diagram illustrates the formal definitions of the Ethereum yellow paper [6] in a non-formal visual way [19]. It gives an almost complete picture of all parts of Ethereum, from mining and the proof-of-work algorithm over a description of the block contents to the Ethereum Virtual Machine (EVM) execution cycle and the internal composition of the different trie data structures. The diagram serves as a good starting point when learning about Ethereum, but it contains some minor inaccuracies, e.g. the so-called substate is missing the list of touched accounts attribute (see section 4.3.2). Unfortunately, some of the descriptions are not self-explanatory without the yellow paper. A more scientific model is EthOn, which is an Ethereum ontology, i.e. a formalization of concepts and relations within the domain [20]. EthOn gives short definitions for all terms related to the Ethereum ecosystem and brings them into relation with each other. These definitions are the basis of several diagrams, which help to visualize certain aspects of Ethereum. There is an overview diagram, as well as more specific charts that model blocks, accounts, transactions, and more. This ontology is very well suited to learn how Ethereum works and to better understand the yellow paper.

Many scientific research papers about Ethereum deal with clustering or labeling of addresses or contracts. In "*Characterizing the ethereum address space*" [21], Payette et al. apply hierarchical clustering, Birch clustering, and the k-means clustering algorithm to the Ethereum address space. Using the k-means algorithm, they group the address space into four clusters. The best number of clusters is determined using the Calinski-Harabasz score, which is higher if the clusters are more distinct. The identified clusters differ in several properties, like the average amount of Ether an account holds, how many outgoing transactions there are, and more. The work by Norvill et al. [22] goes in a similar direction, but they use the affinity propagation and k-medoids clustering algorithms. They download verified contracts from Etherscan and label them depending on keywords that occur in the source code. They identify seven clusters, mostly related to the so-called DAO (decentralized autonomous organization) and gambling.

Several publications have been made about Ponzi schemes. A Ponzi scheme is a type of investment fraud that promises high return on investment rates, which rely on funds that come from future investors. This scheme can only be maintained as long as new investors can be acquired. When no new investors take part in the fraud, the Ponzi scheme inevitably collapses. Ponzi schemes can be implemented on the blockchain using smart contracts. In their paper *"Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact"* [23] Bartoletti et al. manually collect 137 Ponzi scheme smart contracts on the Ethereum blockchain from various sources and then search the rest of the blockchain for similar contracts. They identify four categories of Ponzi schemes and measure statistics, like the economic impact, gains and losses, and the lifetimes of these schemes. Chen et al. [24] build on the work by Bartoletti et al. and create a classification model that detects Ponzi schemes on the Ethereum blockchain in the moment of their creation. They estimate that there were more than 400 Ponzi schemes on the Ethereum platform in May 2017. The classification is based on specific characteristics, like Ether flow and distribution of the opcodes in the contract.

Other research papers in the field present some statistics about Ethereum. In *"Towards analyzing the complexity landscape of solidity based ethereum smart contracts"* [25], Hegedus downloads verified smart contracts from Etherscan and applies some object-oriented metrics to the source codes written in the contract-programming language Solidity. Among the examined metrics are the number of source lines in the contract, number of functions, weighted complexity of functions, and more. Hegedus concludes that smart contracts are short and not very complex. In their paper *"Analyzing Ethereum's Contract Topology"* [26], Kiffer et al. go a step further and modify the Geth client to obtain statistics about all smart contracts that are deployed on the Ethereum blockchain. They state that there are almost three times more smart contracts created by other contracts than by externally owned accounts. In addition, their study reveals that only 10% of user-created and 1% of contract-created contracts are unique.

A popular research field is the development of data extraction and analysis platforms for Ethereum. The *"Ethereum Query Language" (EQL)* developed by Bragagnolo et al. [27] allows users to obtain information directly from the Ethereum blockchain using SQL-like queries. It is possible to query specific attributes from blocks, transactions, accounts, and contracts that can be further confined using where- and order-by-clauses. *"Smashing ethereum smart contracts for fun and real profit"* [28] introduces the security analysis tool *"Mythrill"* for Ethereum. Using symbolic execution, static analysis, and control-flow-checking, the tool is capable of detecting security vulnerabilities of smart contracts. Zhou et al. present *"Erays"* [29], a smart contract reverse engineering tool. It produces high-level pseudocode from contract bytecode for manual analysis making it easy to investigate different contract properties, like McCabe code complexity and code reuse. Also, contracts with no previously available Solidity code can be linked to verified contracts using Erays. Another security analysis framework for Ethereum smart contracts is *"Vandal"* by Brent et al. [30]. It converts bytecode to semantic logic relations and allows users to write own security analyses in the declarative language Soufflé. These logical specifications are then transformed into high-performance C++ code, which can be run against all Ethereum smart contracts to check for contracts where the vulnerability is present. The authors claim that their tool is faster than other state of the art analysis tools, like Mythrill.

In order to prevent bugs, one approach is to apply formal verification to smart contracts. Bhargavan et al. [31] use the functional programming language F* to verify both the runtime safety and the functional correctness of smart contracts. They consider a subset of Solidity and convert both programs of that subset and the bytecode output produced by the Solidity compiler to F* in order to verify that the two are equivalent. Hirai [32] formally defines the Ethereum Virtual Machine in Lem, which is an intermediate language that can be translated into interactive theorem provers like Isabelle/HOL. The author then uses these definitions to prove several properties of smart contracts, e.g. that a program always fails on reentrance, which is an important safety measure.

Finally, Chan and Olmsted [33] analyze the Ethereum transaction graph by organizing all transactions in a graph database. They manage to track transactions from one specific hack that ended up in known addresses of cryptocurrency exchanges.

3.2 Bitcoin and Other Blockchains

Publications about Bitcoin and other blockchains are also relevant for our research, as some approaches might be applicable to Ethereum as well. In *"Quantitative analysis of the full bitcoin transaction graph"* [34], Ron and Shamir describe how they use Bitcoin transactions with multiple sending addresses to cluster addresses into entities. They argue that it is very likely that all sending addresses of a single transaction belong to the same owner and then compute the transitive closure over the sending addresses of all transactions. The resulting transaction graph over these entities is much more meaningful than over just addresses. Then, some statistical properties about the transaction graph are presented, e.g. the amount of Bitcoin they hold. Fleder et al. [35] go a step further in their analysis of the transaction graph and additionally try to assign entities to real people by collecting addresses posted in online forums. Furthermore, they apply the PageRank algorithm to find central entities.

Another research field of interest is anomaly detection in blockchains. Pham and Lee apply k-means clustering, Mahalanobis distance, and unsupervised support vector machine to a subset of the Bitcoin transaction graph in *"Anomaly detection in bitcoin network using unsupervised learning methods"* [36]. Using these three methods, they manage to detect several instances of stolen Bitcoins in the network. Bogner's paper *"Seeing is understanding: anomaly detection in blockchains with visualized features"* [37] presents the system they built for monitoring the Ethereum blockchain for anomalies. The live system generates alerts when it detects an anomaly that can then be further investigated by an expert. A dashboard with relevant information and statistics is presented to the user to assess the severity of the anomaly.

Kalodner et al. develop the blockchain analytics platform *BlockSci* [38]. The tool separates parsing blockchain data and analysis from each other, making it possible to use data from many blockchains that have a similar format as Bitcoin, e.g. Litecoin, Dash, and ZCash. However, account-based platforms like Ethereum are not supported.

BlockSci analyses can be written in Python and C++, and in the paper several analysis applications are shown. For example, they investigate why some miners do not construct blocks that maximize their revenues through transaction fees.

3.3 Reverse Engineering

As smart contracts on the blockchain are only available in bytecode form, reverse engineering techniques need to be applied in order to understand the functionality of the code. While there are not many research papers about reverse engineering smart contracts, blog posts are a good source of information for this topic. Arvanaghi's two-part post [39] gives a good introduction on how to read EVM bytecode. He explains what the most essential instructions do by going over a sample contract, thereby explaining where to find initialization, function dispatching, and self-destruction in the contract bytecode and how exactly each part works.

Sakharov implements an analysis plugin and a debugger for the open-source reverse engineering framework *radare2* [40]. He goes over a simple smart contract instruction by instruction and explains both the functionality of the program and the usage of radare2. The tool helps users to read and understand bytecode by formatting the opcodes, adding instruction offsets, and drawing arrows to where jump instructions jump.

4 The Model

We present our conceptual model for Ethereum blockchain analytics in this chapter. It is a comprehensive representation of the whole Ethereum ecosystem and is split into four parts: Source, EVM, Storage, and Ledger. As our modeling language, we chose UML class diagrams.

4.1 Overview

Before discussing the four aforementioned parts in detail, we first need to explain some general mechanisms and principles of the Ethereum platform.

4.1.1 Account Types in Ethereum

In general, we distinguish between two different types of accounts in Ethereum: externally owned accounts (EOA) and smart contracts [41]. Each account is identified by a 160-bit long address. In the case of an EOA, this address is defined as the rightmost 160 bits of the Keccak-256 hash of an ECDSA public key. The private key of the account is used to sign transactions from that account. A contract account on the other hand cannot make transactions on its own because it is controlled by code and there is no explicit private key known for it. The code of a contract is immutable.

4.1.2 Transactions and Messages

Fundamentally speaking, a transaction is a set of signed data with certain contents that is sent from one externally owned account to another account (EOA or contract) [41]. We explain the exact fields that a transaction consists of further in section 4.5.2, but basically a transaction contains the sender, the recipient, how much money to transfer (if any), and additional data, like which function to call and the arguments.

Contracts cannot initiate transactions on their own, but they can send so-called messages to other accounts if they receive a transaction or a message from another account. Once initiated, a message has the same possibilities as a transaction, but it does not explicitly appear as an entry in a block of the blockchain. It is only implicitly there as a result of a transaction.

4.1.3 Ether and Gas

Ether (ETH) is the currency of Ethereum. It can be divided into 10^{18} parts. The smallest unit is called wei, i.e. 1 ETH = 10^{18} wei. For every transaction and message, the sender can specify the amount of Ether they want to transfer to the recipient.

In order to give miners an incentive to include a transaction in a block, the sender has to spend some transaction fees. Every code instruction in Ethereum is associated with a so-called gas value, which specifies how expensive the instruction is. This value is defined in the yellow paper [6]. For example, a simple `ADD` instruction costs 3 gas, a more complex `SHA3` instruction costs 30 gas, and a `CREATE` costs 32,000 gas. Depending on the executed code, a different amount of gas is consumed. The base fee for a transaction is 21,000 gas, which is the same amount required for executing a normal transaction from an externally owned account to another EOA.

The exchange rate between gas and Ether (which is usually referred to as "gas price") is set by the sender of the transaction themselves. However, if they set it too low, no miner will include it in a block because miners naturally prioritize transactions by their transaction fees. Additionally, the sender specifies a gas limit, which is the maximum amount of gas units that they are willing to pay. When sending a transaction, the sender has to pay the product of gas limit and gas price upfront. If the transaction succeeds and uses less gas than the gas limit, they will get a refund of any remaining gas. But if the gas limit is set too low and the transaction runs out of gas, they will not get a refund of their money. This is a mechanism to protect against denial-of-service attacks against the network where a malicious actor sends a large amount of transactions that all run out of gas eventually, but consume a lot of computing power. If the attacker was getting a refund of their transaction fees, they could just run this attack with practically no costs. Therefore, in the case of such an out-of-gas exception, any modifications made by the transaction are reverted, but the transaction fees are consumed.

4.1.4 Mining and Consensus

Roughly every twelve to fifteen seconds, a new block is generated and appended to the Ethereum blockchain [42]. When a miner manages to mine a new block, they get to transfer a block reward of currently three ETH plus the transaction fees of all the transactions that they included in the block to a beneficiary address of their choosing. At the time of writing, Ethereum uses a proof-of-work consensus algorithm, but it is planned to switch to a proof-of-stake consensus algorithm in the near future [6].

Sometimes two or more nodes manage to mine a new block at the same height. Then the blockchain splits and each node has to decide on their own, which branch to follow. After a while, one branch will be longer than the other one and the shorter branch will be discarded by all nodes. Blocks in the discarded branch are not part of the true blockchain, but they can be included as so-called ommers[1] in another block. In order to reward miners for ommer blocks, the miner of the ommer will still receive a block reward. However, for older ommers, they get lower rewards.

[1]the gender-neutral term for aunt and uncle

4.2 Source

An integral part of the Ethereum system are smart contracts. This is one of many things that sets Ethereum apart from Bitcoin and other cryptocurrencies. Being able to deploy entire programs on the blockchain turned out to be a powerful tool for developers and end users. However, it is tedious to program smart contracts in the bytecode language of the Ethereum Virtual Machine that the yellow paper introduces. This is why several high-level languages exist that compile to EVM code. The most widely used contract-programming language is Solidity.

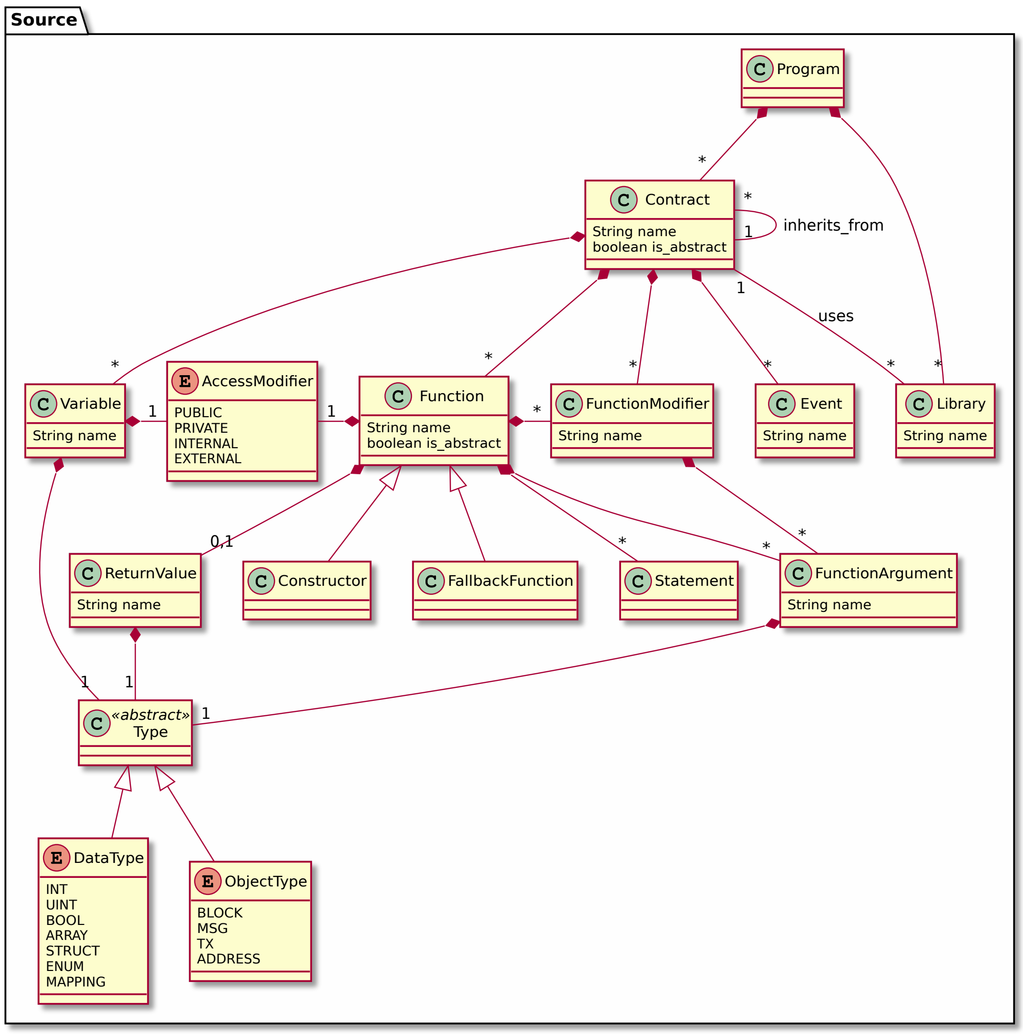

Figure 4.1: Source part of the conceptual model

4.2.1 Solidity

On the official GitHub page, Solidity is described as a "statically typed, contract-oriented, high-level language for implementing smart contracts on the Ethereum platform." [43] It is designed to resemble JavaScript and compiles directly to EVM code, which can be deployed on the blockchain. In the following, we illustrate the different features of the Solidity language with the help of an abstract UML class diagram (figure 4.1) and a concrete contract implementation of a simple auction (figure 4.2).

Structure of a Solidity Program

Every Solidity program consists of any number of contracts and libraries. In the end, only one of the contracts will be deployed on the blockchain, but contracts can inherit from other contracts within the program. A library is a type of contract that does not have any storage and whose functions can be reused by calling them from one or more contracts in the context of the calling contract [44].

Every contract has a name (in our example `SolidityExample`) and consists of any number of variables (line 3 – 7 in figure 4.2), events (line 8), modifiers (line 10 – 13), and functions (line 14 – 47).

A variable consists of a name, type, and access modifier. In Solidity, there are four different access modifiers:

- `public` – Accessible from everywhere
- `private` – Only accessible from inside the current contract
- `internal` – Only accessible from inside the current contract or from child contracts
- `external` – Only accessible from outside the current contract, but not by the contract itself

Functions have a name, an access modifier, statements, and can optionally have function arguments, function modifiers, and a return value. Function modifiers are special properties of functions that are either predefined (like `payable` to indicate that a function can receive Ether) or defined by the developer (like `onlyBy` in our example, which ensures that only a certain address can execute a function).

Two special types of functions exist in Solidity: the constructor (line 14 – 19) and the fallback function (line 47). The constructor is only executed once when the contract is deployed and its code can only be found in the contract creation transaction, but it is not part of the actually deployed code on the blockchain. The fallback function, which has no functionality in our example (and is only included for demonstration purposes), is called when no other function matches the four-byte-identifier of the function. This identifier is also called function signature and is primarily used by the dispatcher.

```
pragma solidity ^0.4.25;
contract SolidityExample {
  address owner;
  bool isRunning;
  uint256 highestBid;
  address highestBidder;
  mapping(address => uint256) oldBids;
  event NewHighestBid(address bidder, uint256 amount);

  modifier onlyBy(address who) {
    require(msg.sender == who, "not authorized");
    _;
  }
  constructor() public {
    owner = msg.sender;
    isRunning = true;
    highestBid = 0;
    highestBidder = 0;
  }
  function placeBid() public payable {
    require(isRunning, "auction has already ended");
    require(msg.value > highestBid, "bid is too low");

    if(highestBid > 0) {
      oldBids[highestBidder] += highestBid;
    }
    highestBid = msg.value;
    highestBidder = msg.sender;
    emit NewHighestBid(highestBidder, highestBid);
  }
  function withdraw() public {
    uint256 amount = oldBids[msg.sender];
    if(amount > 0) {
      msg.sender.transfer(amount);
      oldBids[msg.sender] = 0;
    }
  }
  function getOldBidFor(address bidder) public view returns (uint256) {
    return oldBids[bidder];
  }
  function endAuction() public onlyBy(owner) {
    if(isRunning) {
      isRunning = false;
      owner.transfer(highestBid);
    }
  }
  function() public { }
}
```

Figure 4.2: Example of a simple auction in Solidity

Function Dispatching in Solidity

At the beginning of the bytecode of every compiled Solidity program, there is the function dispatcher, which decides where in the code to jump to in order to execute the correct function that the user intends to call. For every function in the Solidity source code, the Keccak-256 hash of the name and parameter types is calculated and the first four bytes are used as unique identifier. As an example, for the function `getOldBidFor(address` `bidder)`, hashing the string `"getOldBidFor(address)"` results in the four-byte-identifier `0x433978be`.

When making a function call to a Solidity program, the first four bytes of the transaction input data are interpreted as the four-byte-identifier of the function to call. The dispatcher compares against the four-byte-identifiers of all public functions in the code and if it matches one, it jumps to the position where the bytecode for that function is located. If no match is found, but there is a fallback function, it will jump there. Otherwise, it will revert the transaction.

The dispatcher does not perform checks that the amount of parameters and their types of the function that is called is correct. Neither are function modifiers enforced (e.g. with a check that a function that is not `payable` does not receive Ether). These checks take place in the function itself.

Types in Solidity

We distinguish between two different kinds of types in Solidity: data types and object types. The former are fundamental types like `int`, `bool`, but also arrays or mappings between two types (see for example line 7 in figure 4.1 for a mapping between addresses and `uint256`s). The latter are more complex objects with multiple attributes. One example is the `msg` object, which contains information about the current message, like the sender (see line 15) or the amount of transferred Ether (line 27).

Functional Overview of the Auction Contract

Having looked at the structural setup of the auction contract in figure 4.2, we now explain its actual function. When the contract is deployed on the blockchain, the owner is set to the contract creator and the auction is started. This happens in the constructor (line 14 – 19). Now anyone can call the `placeBid()` function (line 20 – 30) and send Ether to the contract. Within the function, it is checked that the auction has not ended yet and that the amount of Ether sent with the transaction is higher than the current highest bid. If both conditions are fulfilled, the account that sent the bid gets set as the new highest bidder and their bid is stored. Additionally, the former highest bid is stored in the mapping `oldBids` and the `NewHighestBid` event is emitted.

If a bid placed by one account is surpassed by another account, the former account can withdraw the amount of Ether that they sent to the contract again by calling the `withdraw()` function (line 31 – 37). If the `oldBids` mapping contains a positive number for the calling account, that amount is transferred to the caller and the value in the mapping is reset. For any account, the current amount of Ether that can be withdrawn this way can be seen by calling the `getOldBidFor()` function (line 38 – 40).

At any time, the contract-creator account can call the `endAuction()` function (line 41 – 46), which sets the auction to a halt and transfers the current highest bid to the contract owner. The function modifier `onlyBy(owner)` enforces that this function can only be called by the contract-creator.

The presented contract implements an auction without a trusted third party. It is decentralized and publicly documented. Only the subject that is bid on needs to be transferred off-chain.

4.2.2 Other Languages

The de facto standard language for writing smart contracts for the Ethereum platform is Solidity. However, there are more languages compiling to EVM code. One noteworthy example is LLL, which "is a low level language similar to Assembly" [45]. It is specifically designed to be simple and not feature-packed like Solidity. However, for developers who want to use a high-level language to write smart contracts, LLL is not an adequate choice because code written in it resembles assembly so much.

Development for LLL started at around the same time as for Solidity, but in the end, Solidity prevailed. Nowadays it is hard to find even a single smart contract that is not written in Solidity. This is why we only consider Solidity in this thesis.

4.3 EVM

The Ethereum Virtual Machine (EVM) is a stack-based architecture with 256-bit words. It is responsible for executing the bytecode that is produced by the Solidity compiler. All information in this section is derived from the Ethereum yellow paper [6] and all explanations are based on the class diagram in figure 4.3.

4.3.1 Execution Cycle of EVM Code

When a miner creates a new block they have to execute all the transactions one after another in the transactions list of that block. Any other node has to do the same thing when they receive a new block from the network. Every node has to execute every transaction and keep track of the entire state of the environment. Responsible for executing a single transaction is the iterator function (see section 9.5 in the yellow paper). It retrieves the next instruction from the contract bytecode, updates the stack, subtracts the gas used for that instruction from the max gas value, and finally increments the program counter (except if the current instruction is a jump instruction, then the program counter is set to the jump destination). That cycle is repeated until the program halts.

The execution environment contains important properties of the currently executed transaction (see section 9.3 in the yellow paper). That includes the code owner, the sender of the transaction, the gas price, the input data, the address that caused the invocation of the code (which is only different from the sender in case of a message call), the Ether value passed with this transaction, the message call depth, the current block header, and the machine code to be executed.

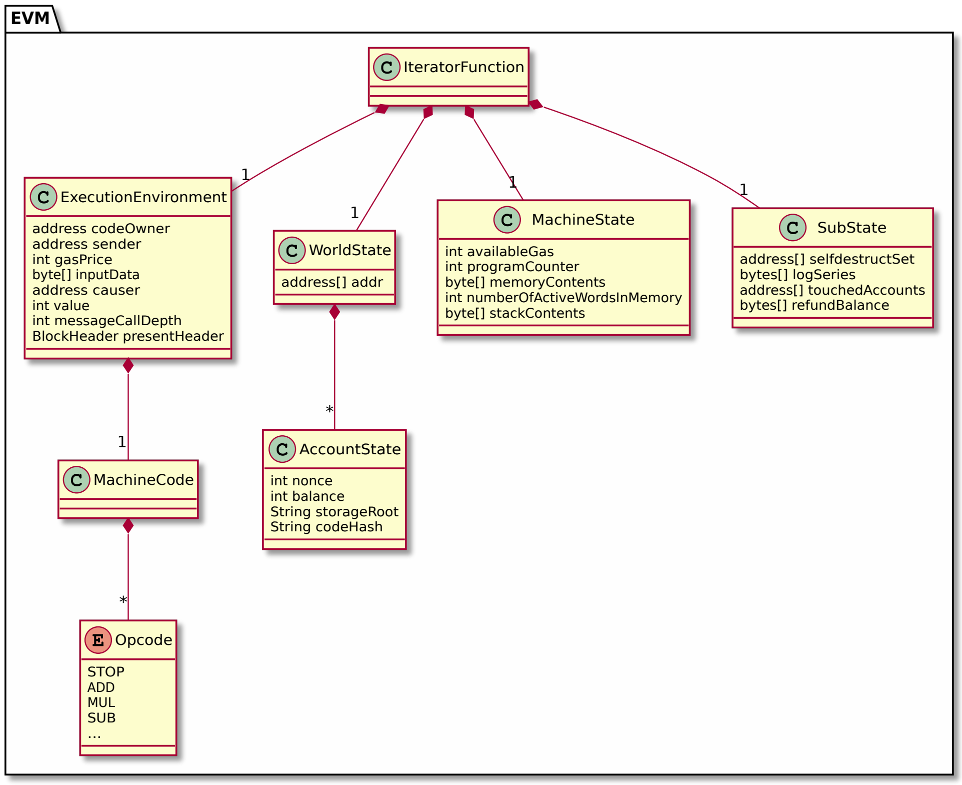

Figure 4.3: EVM part of the conceptual model

4.3.2 States

Every node in the Ethereum network has to keep track of the entire state of the environment. The state is split into four parts.

World State and Account State

The world state maps addresses to account states (see section 4.1 in the yellow paper). For every account (whether it be an externally owned account or a contract), every node in the network has to keep track of its account state, consisting of the following attributes:

- **nonce** – For externally owned accounts the nonce represents the number of transactions that were made from this account. If the account is a smart contract, this is the number of contracts that this contract has created so far.
- **balance** – How much Ether this account has.
- **storage root** – The root of a Merkle Patricia tree that contains the entire storage of this account. In case of a contract that are for example global variables. We explain how storage in Ethereum works in the next section.

- **code hash** – The code hash associated with this account is used to easily look up the actual code and to not have to store code twice if there are multiple contracts with the same code.

Machine State

While the world state and account state are continuously updated with every processed transaction (also across blocks), the machine state (see section 9.4.1 in the yellow paper) is re-initialized for every new transaction. It contains these variables of the currently executing transaction: how much gas is left, the program counter, memory contents, how many words in memory are active, and the stack contents.

Substate

Finally, there is the substate (see section 6.1 in the yellow paper), which contains information for after the transaction is fully executed. The self-destruct set is a list of accounts that will be discarded after the transaction, e.g. because a contract executed the `SELFDESTRUCT` opcode. Then there is the log series, which is a list of logs that arose during the transaction and the list of touched accounts, which is necessary in order to delete empty ones after the transaction. If contract storage is freed during a transaction, the refund balance is increased. This can only be caused by the `SSTORE` instruction and only when the storage is set to zero from a non-zero value. When that happens, the total transaction execution costs are decreased in the end.

4.4 Storage

As seen in the previous section, every node in the Ethereum network has to maintain a database of the world state, containing all account states. In this section, we elaborate on this and on what other data a node needs to store (see figure 4.4).

4.4.1 Tries

Almost all data in the Ethereum system is stored in Merkle Patricia trees (see section 2.1.3). There are four main tries, which nodes of the network have to build up and update as new blocks are generated.

- **State Trie** – First of all, there is the state trie, which is a representation of the world state. It contains all addresses of all accounts and maps them to the account states. Its current root hash is included in the block header of every block.

- **Storage Trie** – For every account, there is one storage trie, which contains the storage contents of this account. Every account state in the world trie contains the root of the storage trie of this account.

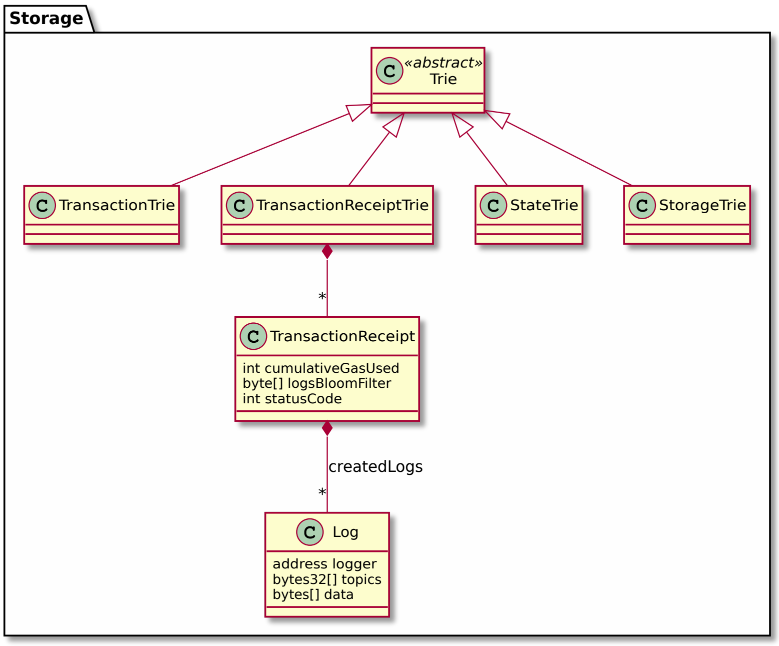

Figure 4.4: Storage part of the conceptual model

- **Transaction Trie** – With every new block, a new transaction trie is built up, containing all the transactions from the block. The root hash is included in the block header. We elaborate on which elements are part of a transaction in the next section.

- **Transaction Receipt Trie** – The transaction receipt trie is almost exactly the same as the transaction trie, except that it contains the transaction receipts that every transaction produces. Such a receipt (see section 4.3.1 in the yellow paper) contains the cumulative gas that all transactions in the block used, up until the transaction for which the receipt is. Furthermore, a receipt contains the logs that were created by the transaction, the bloom filter derived from these logs, and the transaction's status code.

  Logs, in turn consist of the address that created the log, an array of log topics, and the data of the logs themselves.

4.4.2 Implementation in Geth

Geth uses the database LevelDB[2] in order to store all tries that Ethereum requires. On its official GitHub page, LevelDB is described as "a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values". That

[2] `https://github.com/google/leveldb`

makes it easy to look up a specific key, e.g. a code hash, or to iterate over all accounts by traversing the state trie. However, certain requests, like getting all transactions from one account, would take a very long time as there is no connection in the database between accounts and transactions because all data is only stored in form of separate key-value pairs. For that request, one would have to iterate over all transactions and check if it is from the target account. This is quite cumbersome, especially if one is interested in the transactions of many accounts.

4.5 Ledger

The last part of our conceptual model describing the Ethereum system encompasses everything that is actually on the blockchain itself. A block consists of a block header, a list of transactions, and a list of ommers. Again, most of the information in this section is from the yellow paper [6] and is illustrated by the UML class diagram in figure 4.5.

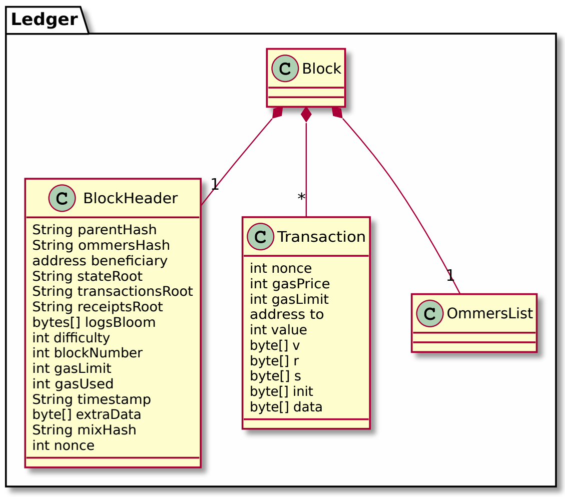

Figure 4.5: Ledger part of the conceptual model

4.5.1 Block Header

These fields are part of the block header of every block (see section 4.3 in the yellow paper):

- **parent hash** – This is the Keccak-256 hash of the header of the parent block.
- **ommers hash** – The Keccak-256 hash of the block's ommers list (see section 4.5.3).
- **beneficiary** – The address that receives the block reward and the transactions fees of this block.

- **state root** – The Keccak-256 hash of the root node of the current state trie.
- **transactions root** – The Keccak-256 hash of the root node of the current transaction trie.
- **receipts root** – The Keccak-256 hash of the root node of the current transaction receipt trie.
- **logs bloom** – The bloom filter of the logs.
- **difficulty** – The difficulty of the block determines how much computation power is needed to find a valid nonce for the block.
- **block number** – A serial number that is incremented for every block.
- **gas limit** – How much gas all transactions in this block may consume at most.
- **gas used** – How much gas the transactions in this block actually used.
- **timestamp** – A timestamp when the block was mined.
- **extra data** – An additional data field that the miner of the block is free to set however they want. Some miners use this field to disclose their name or organization to others in the network.
- **mix hash** – A hash value that satisfies the difficulty requirements together with the nonce.
- **nonce** – A nonce that satisfies the difficulty requirements together with the mix hash.

4.5.2 Transactions

A transaction (see section 4.2 in the yellow paper) consists of the following fields:

- **nonce** – A running number that is increased for every transaction that the sender sends.
- **gas price** – The price in Ether that the sender is willing to pay per gas unit used.
- **gas limit** – How much gas this transaction may consume at most.
- **to** – The address of the recipient.
- **value** – The amount of Ether that is sent together with this transaction.
- **v, r, s** – These three values stem from the signature of the transaction and they can be used to determine the sender.

- **init** – This field is only available for contract-creation transactions. It contains EVM code that initializes the new contract, i.e. it runs the constructor and returns the contract body that is executed every time there is a call to the new contract.
- **data** – This field is only available for message call transactions and contains input data for the called contract, i.e. the function to call and its arguments.

Most blockchain explorers do not distinguish between `init` and `data` because they cannot appear together, so they list it just as one field. But for our model, it is important to be as exact as possible.

4.5.3 Ommers

The third part of a block is a list of ommers. That list contains the block headers of the ommer blocks, but not the transactions. Only by being part of an ommer block, a transaction is not executed, however. A transaction in an ommer block is only executed if it is included in a normal block.

4.6 Correlations Between the Four Parts

With the in-depth description of the four large parts of the Ethereum platform, we answer research question 1 (see section 1.2). These parts cannot be discussed completely separated from each other, as elements of one part have strong connections to another part.

One obvious reference is the translation of a Solidity source code program to machine code. This translation is done by the compiler and during it, our model transitions from Source to EVM.

Also, we can see a direct mapping between the EVM part and the Storage part. The yellow paper demands that every node builds up a world state, which is a mapping between addresses and account states. It even suggests a concrete implementation for this abstract data structure: Merkle Patricia trees. That is why the state trie exactly matches the world state and the storage trie represents the account state.

Finally, the contents of a block, which are modeled in the Ledger part, can be found throughout the entire model: the execution environment contains the complete present block header, the transaction trie contains transactions with all their attributes, and a Solidity program lets the user access certain values from the ledger using the objects `msg`, `tx`, and `block`.

In conclusion, the parts of this complex system should not be looked at on their own, but seen as a whole because they are so closely interwoven with each other. For example, besides understanding how their language works, Solidity programmers should have some knowledge about how the underlying EVM works, since this is how their code is actually executed in the end. Similarly, for end users who are using a web frontend of some distributed application that runs on top of a smart contract, it will be beneficial to look at the source code and see how the smart contract actually works. These web frontends often try to encapsulate the users (who might only have a very basic understanding of distributed ledger technologies) from the actual code to make the user experience as non-technical as possible. But by looking at the Solidity code, users can see if it actually does what the application promises, which is the entire benefit of making an application distributed.

5 Data Acquisition

This chapter deals with research question 2 (see section 1.2), i.e. what data can be extracted from the blockchain and how this can be done efficiently.

5.1 Available Data

Our conceptual model from chapter 4 gives us a very good overview of what data can be extracted from the blockchain. Besides the obvious block headers, transactions, and ommers that are directly part of the blocks of the blockchain, we can also look at the four different tries from the Ethereum client to get data from. Furthermore, there is temporary data that is only available temporarily unless it is stored elsewhere before it is deleted by the Ethereum client. And lastly, off-chain data can be consulted as well.

5.1.1 Blockchain Data

As in every blockchain, blocks are the most fundamental source of data. Roughly every fifteen seconds a new block is mined in Ethereum, consisting of a block header, a transaction list, and an ommer list. The content of these three fields are described in detail in section 4.5.

5.1.2 State Data

Every Ethereum client maintains an internal state from which we can extract data. This internal state encompasses the state trie, storage tries of every account, transaction trie, and transaction receipt trie that were described in section 4.4.1.

In order to read the state data, the data structure of the client has to be read. How this is done exactly follows in section 5.2.2.

5.1.3 Temporary Data

The tries of all Ethereum clients must be synchronized with each other in order for them to represent one single world state. For every new block, each node updates their tries. During that update process, data may be deleted. This is especially true for smart contracts that execute the `SELFDESTRUCT` opcode. A killed contract cannot be used anymore, so there is no need for a client to keep the code or storage of that contract any longer and it deletes the data. However, for analysis of historical data, these terminated contracts might still be of interest.

Another piece of temporary information is data from the mempool. Each node stores new transactions that it receives in temporary storage, called the mempool. When mining a new block, it selects pending transactions from the cache and puts them in the transaction list of the new block. If the node receives a new block from another node, it deletes all transactions from the mempool that are part of that block. The transactions in the mempool do not contain information that is later not directly added to the block. However, metadata information from the mempool might be of interest, e.g. how long a transaction stayed in there. Also, this data can be used to detect front-running (see section 6.2.1).

5.1.4 Off-chain Data

A variety of other data sources exist that are not on the blockchain itself, but can be useful nevertheless. One example would be additional data, which blockchain explorers like Etherscan provide. On Etherscan, users can upload the Solidity source code for their compiled smart contracts and the platform verifies that the uploaded source code does indeed correspond to the bytecode that is on the blockchain[1]. For this to work, the user has to enter the exact compiler version and specify whether optimization was turned on. One limitation, however, is that contract verification is not supported for contract-created contracts.

Public Code Repositories

Another helpful resource are public code repositories, like on GitHub[2]. We can get information about function names that are used in actual Solidity programs from there. That is useful when we want to analyze function signature hashes from bytecode on the blockchain. When calling a public function, that function and its argument types are hashed using the Keccak-256 hash function. Then the first four bytes of that hash make up the function identifier, which the dispatcher at the beginning of the program uses to determine where the code execution needs to jump to for that function. So, in the bytecode of every smart contract on the blockchain, we can find the four-byte hash of each public function. Of course, it is impossible to calculate the function name only based on the hash, but it is possible to create a lookup table for possible function names. In order to generate such a table, real function names from Solidity programs are needed. GitHub and other code repositories can provide us with thousands of different function names.

Libraries

Many contracts make use of helper functions from public libraries. Of special interest are libraries with internal functions because "code of internal library functions and all functions called from therein will at compile time be pulled into the calling contract, and

[1] `https://etherscan.io/verifyContract2`
[2] `https://github.com/`

a regular `JUMP` call will be used instead of a `DELEGATECALL`" [44]. That means that the usage of such a library can be directly seen in the bytecode of a contract.

The most popular Solidity library is the SafeMath library from OpenZeppelin, which provides safe arithmetic operations that cannot over- or underflow [46]. All of its functions are internal. We analyze the usage of the SafeMath library in section 6.6.

5.2 Data Extraction

There are several ways to obtain data from the Ethereum blockchain that we present in this section.

5.2.1 Existing Tools

Several frameworks exist to extract data from the Ethereum blockchain. However, most of them only consider block data and they cannot be used to extract state data at all.

Ethereum ETL

Ethereum ETL[3] (short for extract, transform, and load) is a third-party framework providing easy-to-use scripts to export large amounts of blocks, transactions, and tokens into csv files. It is written in Python and makes use of the Web3 framework (see section 5.2.3). In an article, the author demonstrates its usage [47].

However, exporting the data to csv files, only to import them into an SQL database later on, seems like an unnecessary intermediary step that can be skipped by directly adding the block data into the database.

New Kids on the Block

For their paper *"New kids on the block: an analysis of modern blockchains"* [48], Anderson et al. develop an Ethereum data extractor in NodeJS. Their implementation extracts data directly from Geth's data structure on disk and stores it in a PostgreSQL database. Then, they use that data to perform a security analysis of smart contracts and to generate several statistics, like the lifespan of selfdestructed contracts. However, it is not certain whether this implementation can cope with the current blockchain size, as the paper is from 2016 and they only extract data until block 1,358,548.

Additionally, Anderson et al. implement a modified version of the Geth client in order to find out how many peers there are in the network. This crawler recursively asks other nodes for their peers through the node discovery protocol. Besides estimating the network size, they are able to generate a geographic distribution map based on the IP addresses of the nodes. The code for both tools is available on GitHub[4].

[3] `https://github.com/medvedev1088/ethereum-etl`

[4] `https://github.com/modernblockchains/newkidsontheblock`

Using an API

If the user does not have access to a node, an external API can be used. The famous blockchain explorer website Etherscan also provides an API[5] through which all blockchain data can be accessed from a program. However, it restricts the user to only five requests per second, which is not enough when analyzing hundreds of millions of transactions.

Another popular provider of an API is Infura[6], which does not have any rate limits. But using a centralized API defeats the whole purpose of having a decentralized network. Using the API means having to trust the API provider that the data was not tampered with. Furthermore, as processing data from the entire blockchain would take a very large amount of network requests, we set up our own Ethereum node and exclusively use IPC calls to get blockchain data.

5.2.2 Reading the State Trie

The Ethereum client has its own database where it stores blockchain data and information about the current state. Naturally, an intuitive approach is to read that existing database, which the client builds up itself, directly and use that for data analysis.

A simple proof-of-concept implementation of this approach can be found in the `read_state_trie` sub-directory of the code repository accompanying this thesis.

LevelDB

As mentioned in section 4.4.2, Geth uses the key-value database LevelDB to store information in a dictionary-like data structure. A full node stores the current state trie in the LevelDB, which can be found in the `.ethereum/geth/chaindata` directory.

Several articles describe how Ethereum's database can be read directly [49, 50]. It is important to note that in order to read the state trie for a certain block, the node has to be synced fully up until exactly that block, i.e. it must have downloaded all block data and executed all transactions. If that is given, one can simply look up the state root field in the LevelDB to get binary data that can be parsed into the complete state trie. That trie can now be traversed and all entries can be read successively.

Application

Looking at our model (see section 4.3.2), we see that the elements of the trie contain the following RLP-encoded data: nonce, balance, storage root, and code hash. If we are interested in the code of all smart contracts currently deployed on the blockchain, we can simply traverse the trie and check for every account if the code hash is the hash of the empty string. If that is the case, we found an externally owned account. Otherwise, we found a smart contract and we can take the code hash as the key and look up the actual code for that hash in the LevelDB.

[5] `https://etherscan.io/apis`
[6] `https://infura.io/`

Limitations

One downside with this approach is that smart contracts that executed the `SELFDESTRUCT` opcode will not be found as their code is immediately deleted by the node in order to save disk space. To get all contracts, even those that are deleted at some point, the state trie would have to be read for every single block of the past. Even then, this approach would fail for contracts that are created and destroyed in the same transaction. In section 6.2.2, we show that this is not just a theoretical concept.

Performance is also a limiting factor. We have to go over all accounts in order to find all smart contracts. That includes all externally owned accounts as well, of which there are a lot more than smart contracts.

But the biggest limitation of this approach is that it does not really work with a live node because the state trie is constantly changing when the node is getting new blocks. Therefore, the node's synchronization would have to be halted at a certain block so that the contracts can be read. However, if at a later point synchronization is continued up until a higher block, there is no way to import all the new smart contracts without reevaluating all accounts.

5.2.3 Reading Block Data

In order to read block data from a node, that node must accept either RPC or IPC calls. The difference is that remote procedure calls (RPC) can be sent from another computer via the network and inter process communication (IPC) can only take place between processes on the same machine. Because we do not want to have to create the network requests directly from our program, we use a framework for that.

By making many consecutive IPC calls to the node, we can read all block data and save all information, which is relevant for us in a database. More specifically, we can get user-created smart contracts by looking at transactions with recipient `null`. The address of the contract that is created this way can be calculated deterministically and is only depending on the sender address and the nonce, which are both part of the transaction. The exact formula is described in the yellow paper in section seven [6]. We then query our Geth node with an `eth.getCode()` request to obtain the code that is actually deployed at that address. However, if the `SELFDESTRUCT` instruction has been executed for that contract in the time between deployment and our data collection, `eth.getCode()` returns an empty contract. In that case, we look at the `input` field of the contract-creation transaction, which contains EVM code that runs the constructor and puts the final contract code on the blockchain. We use a simple heuristic to separate contract-creation code and constructor arguments from the actual smart contract bytecode: the contract-creation code starts at the very beginning of the `input` field and the actual bytecode starts afterwards at the first `PUSH1` instruction that pushes a value $\geq 0x60$ on the stack and is located after the first `CODECOPY` instruction. The constructor arguments start after the last `STOP` instruction in the `input` after which there is an unknown operation (or simply after the last `STOP` of the code if there is no unknown operation thereafter). If this heuristic does not match anything or if the complete input does not start with a `PUSH1` instruction, we simply store the complete `input` field as the contract code. We choose this heuristic because smart contracts created by the Solidity compiler generally follow it and we cannot easily predict actual bytecodes of custom contract-creation inputs. Again, this heuristic is only applied for contracts that selfdestructed already.

This way, we only get smart contracts that are created by externally owned accounts. Finding contracts that are created by other smart contracts requires a different strategy as transactions that ultimately lead to a contract creating another contract can only be identified by observing the entire transaction trace, which requires a lot of resources. However, the address at which a contract-created contract is located is also deterministic and only depending on the parent address and its current nonce. For a smart contract, the nonce only increases if it creates a new smart contract. This is contrary to externally owned accounts where the nonce also increases for every other transaction. Also, we only have to consider smart contracts that contain the `CREATE` instruction, which is the only instruction that can create another contract. Therefore, we iterate over all user-created contracts containing a `CREATE` instruction and calculate the address of the contract it would create for the first nonce. Then we call `eth.getCode()` to see if there actually is a contract code at the location. If a contract is found, we store it in our database. We increment the nonce as long as we find new contracts. When there is no contract for a certain nonce, we additionally check 500 more nonces so that if a few contracts in between were deleted already, we still find contracts that were created later but are not deleted yet. This method lacks to get the code of contract-created contracts that selfdestructed.

We define that contracts that are created by an externally owned account are generation zero. A contract that is created by a contract of generation n is generation $n + 1$. After checking all user-created contracts in generation zero for potential child contracts, this process is repeated for all contract-created contracts in higher generations until no further contract is found. Due to the immense amount of contracts in the first generation we set the number of nonces to check after `eth.getCode()` does not find a contract anymore to 50 for all higher generations.

Web3 and PyEthereum

For this data extraction approach we make heavy use of two frameworks. The Web3 framework[7] implements Ethereum's JSON RPC specification. It can be very easily used to make RPC or IPC calls from Python or JavaScript code. Using these calls, we can get all kinds of information from an Ethereum node, like the current block number, specific contents of a block, and much more. Figure 5.1 shows a sample IPC call to the `eth_getBlockByNumber` endpoint. The framework is developed directly by the Ethereum foundation.

The PyEthereum framework[8] complements Web3 with useful tools. It is also maintained by the Ethereum foundation and it contains e.g. cryptographic functions, conversion

[7]`https://github.com/ethereum/web3.py`
[8]`https://github.com/ethereum/pyethereum`

```python
#!/usr/bin/python3
# coding=utf-8

from web3 import IPCProvider

class Geth():
    def __init__(self, ipc_path):
        self.provider = IPCProvider(ipc_path, timeout=60)

    def get_block(self, block_num):
        data = self.provider.make_request("eth_getBlockByNumber", [hex(block_num), True])
        return data["result"]

    def get_blocks(self, start, end):
        for n in range(start, end):
            yield self.get_block(n)
```

Figure 5.1: IPC call in Python using Web3

functions, address manipulation functions, and more.

5.3 Data Storage

Step number two of the data acquisition part is the storage of the data. We evaluated two different database setups, of which the relational database suited our needs.

5.3.1 Graph Database

At first we tried using the graph database Neo4j[9], which describes itself as “The #1 Platform for Connected Data”. A graph seemed like a good structure for our data, as transactions and accounts are heavily connected with each other.

Database Setup

There is an existing repository on GitHub that stores blocks, transactions, and addresses in a Neo4j database[10]. We forked it and made it work with IPC calls instead of only RPC. Transactions and accounts are stored as nodes in a graph and each transaction is connected with one edge to its sender and one to its recipient.

The setup allows us to use Neo4j’s graph query language “Cypher” to obtain data in an intuitive way. For example, the following query returns the contract creator, nonce, and code for all smart contracts:

```
MATCH (sender:Address)<-[:TX_FROM]-(tx:Transaction)-[:TX_TO]->(rcv:Address)
WHERE rcv.hash = 0
RETURN sender.hash, tx.nonce, tx.input;
```

[9] https://neo4j.com/
[10] https://github.com/sardinois/eth_graph

Performance

In the end, we decided against using a graph database because it has performance issues in comparison to a relational database. Our performance testing was twofold: first, we evaluated how long it takes to import blocks, and then we timed retrieval queries on the data structure. The test program to import approximately 280,000 recent blocks finished after one whole day. Next, we ran the query from the previous section on that data set to obtain all smart contracts and it finished after more than four minutes. Both of these times are too high for our purposes because processing the entire blockchain would take weeks and therefore the graph database is no viable solution for our problem.

Storing more than 350 million transactions in any database is quite a challenge and we did not try to store every single transaction and account for our second approach, but only the data that we really are interested in, which is smart contract data. When restricting the data to that subset, it is not really that interconnected anymore and a graph database is not the preferred data structure. Accounts and the transactions between them make up a good graph, but when only looking at smart contracts and smart contract bytecode, there is no clear graph structure.

The original code repository for the Neo4j Ethereum integration is from September 2017. At that time, there was a lot less data on the Ethereum blockchain and performance was not such a big issue back then. But we were looking for a solution that scales with the growth of Ethereum for the next couple of years, which is why we went for a relational database after all.

5.3.2 Relational Database

Having decided that we do not store all transactions and accounts of the entire Ethereum blockchain, but rather only smart contract data, we build up a MySQL database.

Database Schema

Our database schema consists of the following nine tables:

- `eth.block` – This table simply stores each block with its block number, hash, and timestamp.
- `eth.contractTransaction` – As storing all transactions is infeasible, we only store transactions that create a new contract, i.e. transactions to address 0. All the fields that the Geth client provides for the transaction are columns in the table.
- `eth.contract` – In this table we store all smart contracts that were created by an externally owned account. We store the address, the transaction hash, on which network it was mined (mainnet or one of the test networks), the contract-creation code and the constructor arguments that were passed with the contract-creation transaction, and whether `SELFDESTRUCT` has been executed for the contract. Furthermore, we store the Keccak-256 hash of the contract bytecode, which is a foreign key for the `eth.contractCode` table.

- `eth.contractCode` – In order to save disk space and to improve performance, we do not store duplicate contract bytecodes. Therefore, we store the code in this separate table and reference it using its hash from everywhere where it is used. Additionally, we store whether this bytecode contains a `CREATE` opcode and how many times this exact bytecode is referenced. As the result of our analyses later on, five other fields are introduced: a reference to the `verifiedContracts` table if available, and the minimum and maximum compiler and SafeMath versions that could be determined.
- `eth.contractCreatedContract` – This table is similar to `eth.contract`, though not for contracts that were created by externally owned accounts, but by other contracts. It also references `eth.contractCode` but has additional fields: the creator address, the used nonce, and generation. The generation is $n + 1$ if this contract was created by a contract of generation n and contracts that were created by externally owned accounts are implicitly of generation 0.
- `eth.functions` – The functions table is a lookup table of function signatures and four-byte function signature hashes. It was generated as part of other works at our chair using publicly available function names from Etherscan.
- `eth.compiler` – We use this table to store information about the different versions of the Solidity compilers, including version number, long version string, and release date.
- `eth.library` – Similar to the previous table, this table is used to store information about different versions of libraries, like the SafeMath library. Columns are the library name, version number, release date and the functions that this specific library version implements.
- `eth.libraryFunction` – In this table we store bytecodes of library functions that we were able to extract from the library, together with the compiler version used, whether optimization was turned on, the library name and version used, and of course the function name.
- `verifiedContracts` – We use this table to verify the results of our research. It contains all verified contracts from Etherscan and was kindly provided to us by another member of our chair.

Performance

This database setup, which does not store all transactions and accounts, performs much better than the previous approach. For all our analyses (see chapter 6), we use block data from the first 6,900,000 blocks of the Ethereum blockchain. Going over all blocks and importing all user-created contracts to the relational database took approximately 33 hours. We imported the blocks in chunks of 1,000,000 blocks and figure 5.2 shows the rate of imported blocks and contracts per second for each chunk. Because in the

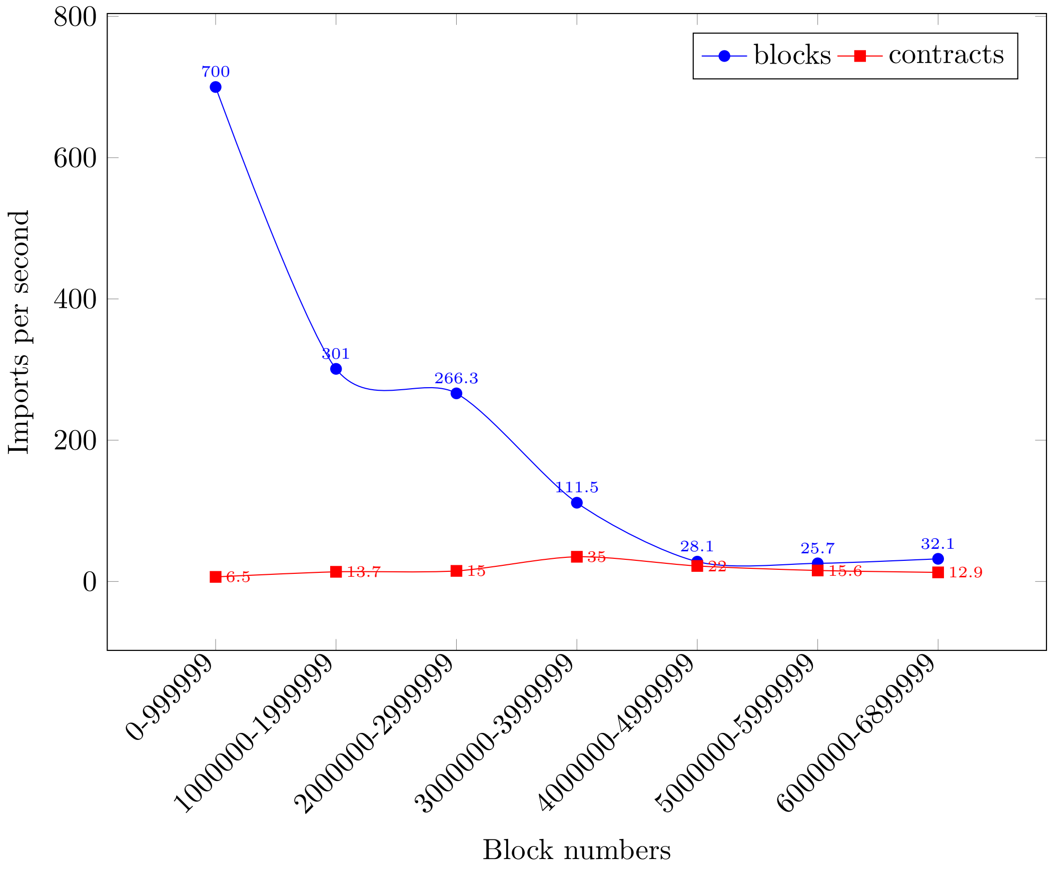

Figure 5.2: Imported blocks per second

first million blocks there are only 9,242 user-created contracts, they took just under 24 minutes to import. In contrast, blocks 5,000,000 to 5,999,999 contain 607,217 contracts and the import ran for over ten hours. In addition, as more and more data is added to the database, importing naturally becomes slower as MySQL has to check for duplicate contract codes and update table indices. With this data structure, going over all bytecodes of all contracts can be done in just a few minutes.

All performance tests for the relational database were run on an Ubuntu 18.04.1 machine with Linux kernel 4.15.0. The system has an *AMD EPYC 7401P* 24-Core 2GHz Processor with 128GB RAM and NVMe storage.

6 Analysis

After describing the Ethereum system in a conceptual model, acquiring data from the blockchain, and storing it efficiently, we are now able to analyze the data. In the following sections, we answer research questions 3, 4, and 5 by analyzing different anomalies in the network, investigating ERC standard token usage, approximating Solidity compiler versions of deployed bytecode, and detecting usage of the *"SafeMath"* library.

6.1 General Information

First, we describe both the data and our code repository that we use for analysis in a general way. This is the basis for our analyses later on.

6.1.1 Underlying Data

As described in the previous chapter, we use the first 6,900,000 blocks of the Ethereum blockchain for our analysis, i.e. block 0 until 6,899,999. The first block after the genesis block was mined on July 30, 2015, and the last block is from December 16, 2018 UTC. All user-created contracts in our database are from that time frame. However, due to our method of obtaining contract-created contracts, it is not guaranteed that these contracts are also exclusively from these blocks. When we discover a child contract of another contract, it is not possible for us to determine on which date it was originally created, because we do not know the transaction that ultimately lead to the creation of the contract. Because importing all contract-created contracts into the database takes several days, a few contracts that were created after block 6,899,999 are also added to our data set. This is not a problem or limitation for our analysis, but should be noted for completeness.

The latest Solidity compiler version that we consider is 0.5.2, which was released on December 19, 2018, and is the 46th official release. For the SafeMath library, the highest version number is 2.1.1 from January 4, 2019. When counting all release candidates (RC) as separate release, there are 31 versions.

Statistics

Next, we present some general statistics of our data set. Table 6.1 shows how many user-created and contract-created contracts there are, and how many contract codes exist. Throughout our analysis in this chapter, we distinguish between contracts and contract codes. A contract code is EVM bytecode that is deployed any number of times on the blockchain. Often, both metrics are relevant. Storing contract codes in a separate database table and referencing them from the `contract` and `contractCreatedContract` table minimizes disk space and maximizes performance. We say a contract code is unique, if there is exactly one single smart contract with that bytecode on the blockchain. In that case, we call the contract unique as well. While 162,630 contract codes are unique, 27,596 are deployed two or more times on the blockchain.

Type	**Total**	**Unique**	**With `CREATE` opcode**
User-created contacts	2,176,953	160,966	143,878
Contract-created contracts	4,803,762	1,664	1,921,588
Contract codes	190,226	162,630	14,901

Table 6.1: Amount of entries in the database

What can be seen in table 6.1 is that more than two thirds of all contracts are created by other smart contracts. Furthermore, only about 7.39% of user-created and 0.03% of contract-created contracts are unique. These results coincide with the ones from Kiffer et al. [26]. Their data set, however only encompasses the first 5,000,000 blocks.

About 30% of all contracts contain a `CREATE` opcode, i.e. they can create other smart contracts themselves. 83,310 (3.8%) user-created contracts have executed the `SELFDESTRUCT` instruction between their initiation and our data collection.

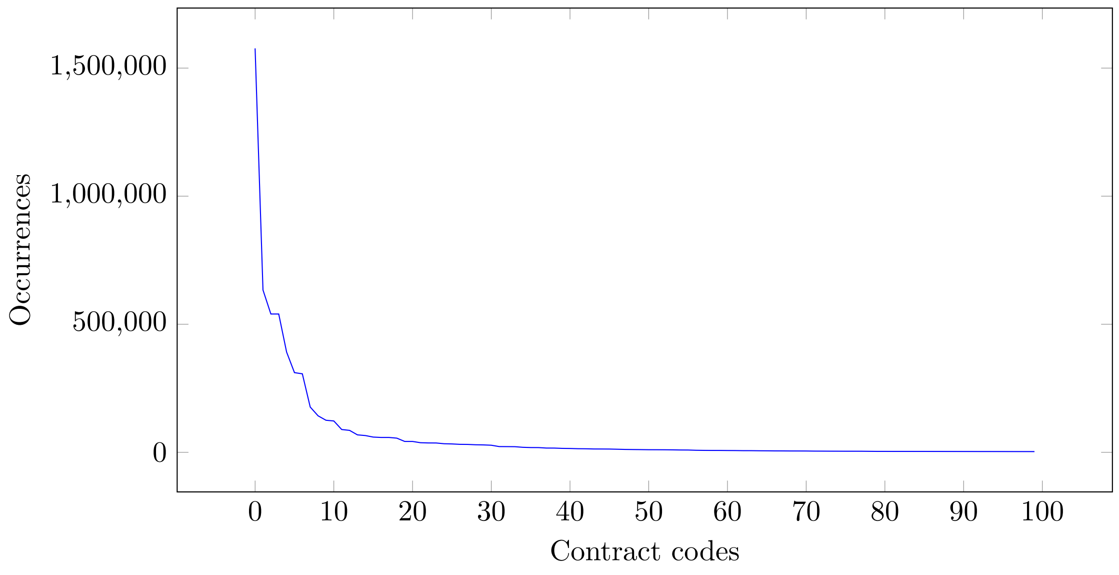

Figure 6.1: Occurrences of the hundred most used contract codes

Figure 6.1 shows for the top one hundred most used contract codes, how often they occur on the blockchain. The distribution follows a power law [51]. The most used contract code is used by 1,577,003 different smart contracts. With further investigation, we find that these contracts are all created by the cryptocurrency exchange Bittrex[1]. The

[1] `https://international.bittrex.com/`

exchange apparently creates one smart contract for each user account to facilitate token management. We go more into detail about the most used smart contract codes in section 6.3.2.

6.1.2 Code Repository Setup

So far, all our results are derived directly from the database using SQL queries. For all our following analyses there is a dedicated Python script, which performs SQL queries on the database and further processes the data, making much more powerful analyses possible than with simple SQL queries.

In the code repository, there are multiple folders with code. The `contract_analysis` folder contains all analysis and evaluation scripts that we present in this and in the next chapter. Except for the two helper scripts, each script can be run independently. When running a script with the `--help` option, a short description of the script is given and the possible command line arguments are shown. Most of the time, the arguments consist of the database username and password.

The folder `database_model` contains two SQL scripts. With `create.sql`, the whole database schema can be created from scratch (see section 5.3.2). It also directly fills the `compiler` and `library` tables. The `functions` table is created by the `functions.sql` script.

Finally, the `multi_compile_contracts` folder contains a Node.js project whose functionality we explain in detail in section 6.6.1. It is written in Node.js because it makes use of *solc-js*[2] in order to compile Solidity programs with older compiler versions. In the directory, there is a folder with all OpenZeppelin SafeMath releases and one with the Emscripten compiled binaries for every compiler version of Solidity.

Helper Scripts

As mentioned, there are two helper scripts in the `contract_analysis` folder that cannot be directly executed but provide common functionality to all other scripts. `opcodes.py` contains only a single variable: a list of all opcodes of the EVM. The list is composed in a way that the text code at a certain index corresponds to the hexadecimal value of the instruction. To obtain the opcode for a specific hexadecimal value, it is sufficient to access the list at that index. The `eth_util.py` file contains several helper and debug functions to facilitate development of our analyses. For example, it provides functionality to convert bytecode to opcodes, to calculate function signature hashes, to compare version numbers of compilers, and more. Each function in the script can each be run independently.

6.2 Anomalies

An anomaly is an unusual event or an entity that behaves abnormally. These are of interest because they give a different insight of what is happening in the network than

[2]`https://github.com/ethereum/solc-js`

by simply looking at normal behavior. In a way, anomaly detection can be seen as the counterpart of clustering. We pursue multiple strategies to find anomalies. The following sections answer research question 5 by showing three very different events that are all unusual in their own way.

6.2.1 Front-running

Front-running is a phenomenon that originally stems from the stock exchange. One party – the front-runner – performs a trade, knowing that a very large trade for the same financial asset by some other party is about to happen shortly. The front-runner places their own trade so that it occurs before the large trade and they benefit from the resulting price change that is induced by the large trade [52].

In Ethereum, front-running is also possible because all transactions are public and before a transaction is actually executed, it has to be distributed in the network. Each full node stores the list of pending transactions in its mempool. Therefore, an attacker can monitor the mempool and if they see a transaction to a token smart contract that transfers a large amount of tokens, they could immediately place their own transaction to the same contract where they buy or sell tokens, depending on how they expect the price to be affected by the pending transaction. In order to actually front-run the transaction, the attacker sets a higher gas price for their transaction so that miners prioritize that transaction over the original one.

Detecting front-running requires an accurate view of the mempool, which is not permanently stored by the full node itself (see section 5.1.3). Therefore, in order to spot such anomalies, either the mempool has to be stored manually for later analysis, or a live detection system needs to be implemented. The script `detect_front_running.py` represents a prototypical implementation of the later approach. It leverages Web3's `eth.getBlock("pending")` call to obtain all transactions in the mempool. New transactions are stored in a temporary list containing all transactions that were seen within the last five minutes. Every transaction in the current mempool is compared against every transaction in that list and we check whether all of the following four conditions are fulfilled:

- The transactions have the same recipient.
- They call the same function.
- One transaction has an at least 50% higher gas price.
- That transaction was issued at least 3 seconds after the other one.

We let this script run for ten hours on September 24, 2018, and were able to detect many instances of possible front-running on the Ethereum blockchain. There were over 153,000 combinations of transactions that fulfilled the above four conditions. Almost half of them were to one single contract: `0x22dccFA39DbE59CD3F27E6531B33B5101bb2A70D`. That contract is the *"Daily Divs Card Game"*, which coincidentally was launched on that exact day. Users can buy virtual playing cards from other users and every time a card transfer happens, the card gets a bit more expensive and dividends are payed out to all card owners. A purchase does not have to be explicitly accepted by the card owner. As long as the buyer sends the correct price in Ether to the `buy(uint _card, address _referrer)` function, the transfer happens. Initially, there are twelve cards, of which the card with ID 0 is most expensive with a starting price of 4 ETH. Possessing an expensive card means getting higher dividends from any card transfer. Therefore, the card with ID 0 is very desirable. We investigate the second purchase of the card 0. After the first purchase, its price is now 4.4 ETH. There are four transactions attempting to buy the card at that price, but only one succeeds. All transactions are in block 6,392,902 and table 6.2 shows for each transaction their gas price and the index within the block. The transaction with the lowest index is executed first. It successfully buys the card and all other transactions revert, because the price of the card is now higher than 4.4 ETH. Clearly, if a buyer sets a higher gas price, their transaction is prioritized by the miner and it takes an earlier position in the block. The successful transaction pays a twice as high gas price than the other transactions.

However, it is not entirely clear if the user that was successful in the end actually supervises the mempool for other transactions to the desired card or if they just blindly set a high gas price, and hoped it would be sufficient.

Transaction hash	Block number	Index	Gas price	Successful?
`0xb965fc839269dd1e...`	6,392,902	0	115,000,000,000 Wei	yes
`0xef8acbc75d5d2b94...`	6,392,902	4	66,000,000,000 Wei	no
`0xdc617d52bab9a2f8...`	6,392,902	14	50,000,000,000 Wei	no
`0x539eb8341793cb3c...`	6,392,902	15	50,000,000,000 Wei	no

Table 6.2: Front-running for the Daily Divs card game

6.2.2 Self-destructing Constructors

Another anomaly, which we investigate are smart contracts whose constructor contains the `SELFDESTRUCT` opcode. The script `find_selfdesctruct_in_constructor.py` finds exactly such contracts. We use our heuristic from section 5.2.3 to detect the constructor. Due to this methodology, the detection does not work for contract-created smart contracts. In total, there are 43 user-created contracts, which immediately end themselves. In contrast to the two million total user-created contracts, this is very insignificant and this phenomenon is very rare on the Ethereum blockchain. Nevertheless, these smart contracts might still be of interest.

One reason for such a behavior could be to hide certain activities on the blockchain, as block explorers like Etherscan do not show the code of contracts that are selfdestructed anymore. For example, for the smart contract at address `0x07Fb462187E24cbF3F440286837D4Fbb713c107c`, there is exactly one transaction, but the code is not available in the state trie of a normal full node anymore. The `eth.getCode()` call for that address merely returns the empty bytecode `0x`. In order to be able to read the code, one has to look at the contract-creation transaction and then convert the bytecodes to opcodes. Apparently, before its destruction, the contract created another contract, namely the one at address `0x4aB53afb23DEf51a624BE15A6dF5187b76f88AfA`. Through the immediate `SELFDESTRUCT`, the creation of the other contract is obfuscated.

Another reason for this behavior might be a simple programming error. The programmer of the smart contract might have accidentally put the self-destruction code in the constructor instead of a dedicated function. And finally, there are sporadic detection errors that stem from our non-perfect heuristic and from the fact that we can only detect *if* a `SELFDESTRUCT` is in the constructor, but not whether it actually is executed, e.g. due to a jump over that instruction.

6.2.3 Transactions to Future Contracts

The last anomaly that we examine are transactions to accounts that are not smart contracts at the time of the transaction, but can become smart contracts at a later point in time. We call these accounts *future contracts*. Transactions to future contracts are possible, when the sender of the transaction knows that the receiving account can later have code associated with it, because some user or contract can create the contract with exactly that address in the future. This information can be known in advance, since the addresses of smart contracts are calculated deterministically, only depending on the creator address and the nonce.

We distinguish between transactions to future contracts that can be created by contracts and those that can be created by externally owned accounts. First, we look at the former. As mentioned, while importing contract-created contracts, we check a few additional nonces after the last one that returns a smart contract code (see section 5.2.3). For each of these nonces, we also leverage the `eth.getBalance()` call to check whether the future contract that can be created with that nonce has non-zero balance. Note that this method does not find future contracts that are already instantiated at the time the script runs. We found two instances of future contract-created contracts. The first column of table 6.3 lists the parent address in the first row and the address of the future contract in the second row. In the other columns, we note the nonce to create the future contract and the balance at the time the future contract was found. At the time of writing, the account in the first line was not a contract yet, whereas the second one was created 126,614 blocks (22 days) after Ether was first transferred to the account. Our script was able to find it, because it ran within these 22 days.

We only have a closer look at the second contract pair because it involves a considerably larger amount of money. By looking up the four-byte function identifiers of the two contracts in our table, we can understand this anomaly better. The function of the parent contract that created the child contract is `createForwarder()`, which suggests that the child contract's task is to forward money to the parent contract. The child contract itself has the functions `balanceOf(address)`, `transfer(address,uint256)`, `flushTokens(address)`, and `flush()`. Apparently, the `flush()` function transfers the

Parent address and future contract	Nonce	Balance
`0x41238ec6b8d6543f018abc672704747e4b8638df` `0xFaF844B92994fc7194AB0e0B27C6AAf494f66C36`	2	0.000559 ETH
`0xbf0c5d82748ed81b5794e59055725579911e3e4e` `0x9C921987501b13899F773836DebbA360E317C7e5`	52,974	2.80853 ETH

Table 6.3: Future contracts that can be created by other contracts

money to the other contract. This convoluted and confusing sequence obfuscates the transactions and makes it harder to track the flow of money in the network.

For transactions to future contracts that can be created by externally owned accounts, we use a different approach. We iterate over every transaction with a volume of at least 10 ETH and check whether the recipient account is a smart contract. If this is the case, we compare the block number of the transaction with the block number at which the contract was created. If the first one is lower than the second one, we found a transaction to a future contract. This approach only finds future contracts after they were instantiated. Also, this technique does not work for contract-created future contracts because we do not know the block number at which a contract-created contract is created. In total, we found 22 transactions to future contracts with a value of at least 10 ETH. There were 26,821,359 transactions transferring at least 10 ETH in the first 6,900,000 blocks, so this phenomenon is extremely rare, but nevertheless existing. For brevity, table 6.4 only shows the biggest such transactions with at least 70 ETH.

Address	Tx block number	Creation block number	Value
`0x332b656504f4EAbB44C8617A42AF37461a34e9dC`	239,687	243,826	5,306.8 ETH
`0x20FeAF3Db3576611b24F239E395651e0fC94b977`	2,141,302	4,726,586	199.96 ETH
`0x61E2E4e348A253f38f23Ee4D5B2aC17183e74c26`	2,154,969	4,726,578	100.0 ETH
`0xc751125099658dB6c4f79996069b2B89DC914727`	2,164,811	4,726,453	168.0 ETH
`0x9611cd35b858f9505515d82b0EeC9A6616fa795E`	3,307,081	4,729,965	172.63 ETH
`0x8eABaF1e1c35cd8E2677edE965D045bf645baF6e`	3,644,181	4,731,860	300.0 ETH
`0x73cCBb3A2665fF16cEd11A7B10a91eA31A783534`	3,834,990	4,726,798	291.12 ETH
`0xb07Ec92F81244F2a84df3A915D4E5c75CDd34346`	5,239,082	5,240,098	70.31 ETH
`0xA60DC0b1DaC08610543E8E68A9961847e0a13470`	6,211,359	6,215,703	73.0 ETH

Table 6.4: Contracts receiving a transaction of 70 ETH or more before their creation

6.3 Areas of Application of the Ethereum Blockchain

In this section, we answer research question 4: *"What are different areas of application of the Ethereum blockchain?"* Our approach is twofold. First, we determine how many smart contracts implement ERC token standards and afterwards we have a closer look at some of the most used contract codes.

6.3.1 Determining ERC Standard Usage

A token system is a sub currency, which is associated with some value [53]. In Ethereum, a token is implemented as a smart contract. There are official token standards that define clear interfaces to allow interactions between tokens. In order to be compliant to the standard, a concrete token contract implementation needs to provide certain functions and events. The two most famous token standards are ERC20 and ERC721. We present usage statistics for both of them. The script `get_contractHashes_with_certain_functions.py` iterates over all contract codes and compares the four-byte function identifiers of the bytecode with the known four-byte identifiers of all functions of the token standards. Similarly, it compares the 32-byte event identifiers with ones of the standard.

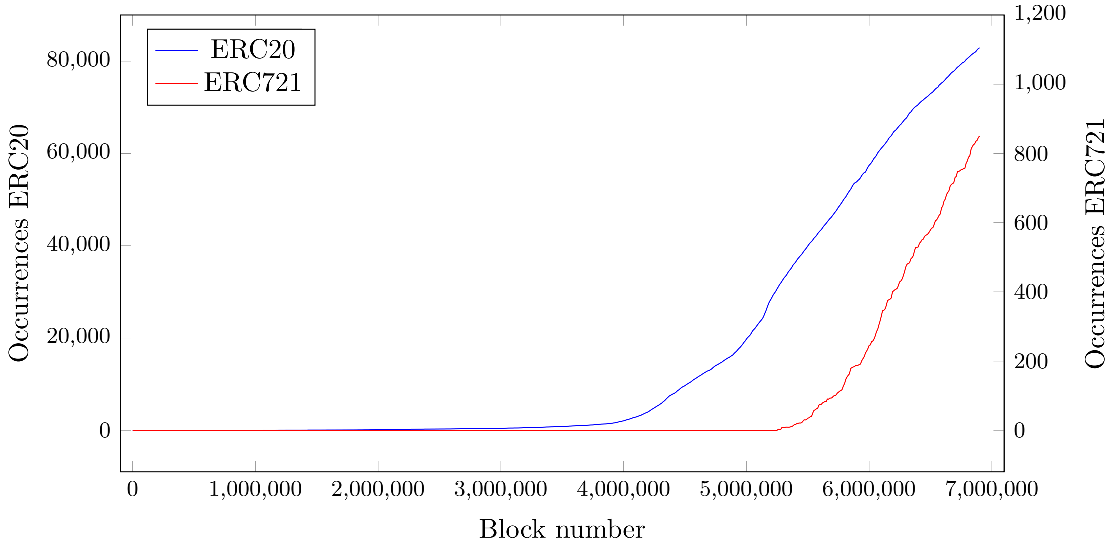

Figure 6.2: Chronological sequence of user-created ERC20 and ERC721 tokens

ERC20 tokens are often used for initial coin offerings (ICOs). The official ERC20 specification contains six required and three optional methods. Additionally, two required events must be implemented [54]. For the non-fungible token standard ERC721, there are nine required methods and three required events [55]. Optional methods exist, but as separate extensions to the standard. For our search for tokens, we only consider the required methods and events. Figure 6.2 shows the chronological sequence for the total number of user-created token contracts for each of the two token standards. Again, we cannot show this kind of data for contract-created contracts because we do not have information of when these contracts were created.

In our data set, there is a total number of 89,326 smart contracts from 51,114 distinct contract codes implementing the ERC20 standard. Of these contracts, the majority were

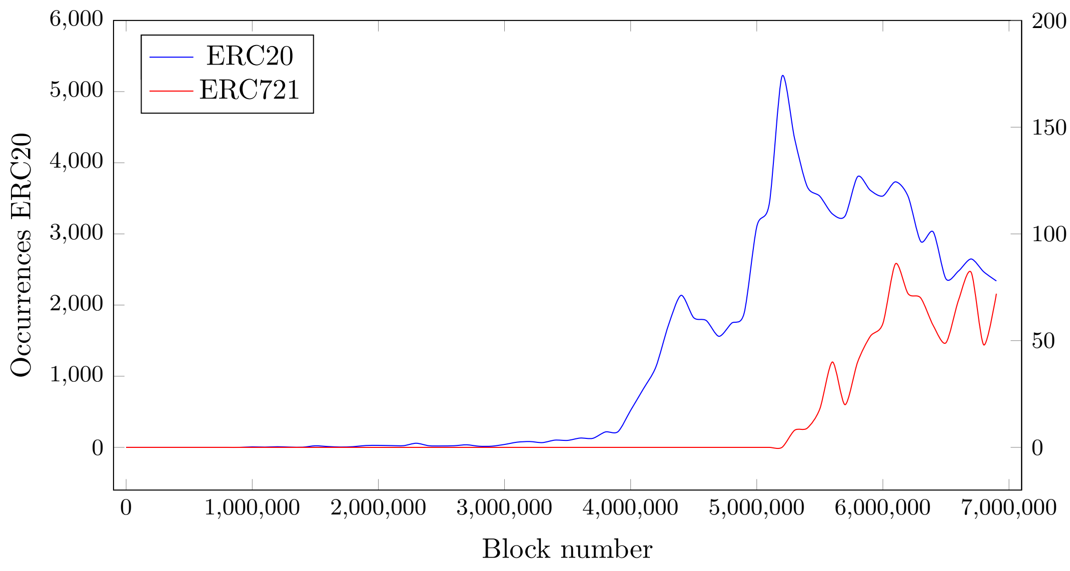

Figure 6.3: Amount of user-created ERC20 and ERC721 tokens per 100,000 blocks

created by externally owned accounts and only 7.2% were created by other contracts. As the ERC721 standard was finalized more recently, there are much fewer contracts implementing it. There are 906 ERC721 tokens from 668 distinct contract codes, of which 6.2% were created by other contracts. The number of contract-created tokens is very low, especially compared to the entirety of smart contracts, where 68.8% of all smart contracts were created by other contracts (see section 6.1.1). Furthermore, only about 1.3% of all smart contracts implement any of these two tokens, but 27.2% of all contract codes. While the code of individual tokens is not copied thousands of times across the blockchain, they make up a substantial amount of the blockchain's code base. This implies that tokens are a popular use case of Ethereum.

In order to get a better understanding of the ERC token standard usage, we look at the rate at which token contracts are published. Figure 6.3 shows how many user-created ERC20 and ERC721 tokens were published on the blockchain for every 100,000 blocks. Between block 5,100,000 and 5,200,000, the publication of ERC20 tokens reached its peak with over 5,000 tokens published. For ERC721, there is no clear peak, but the rate has been more or less constant since block 5,500,000.

6.3.2 Other Applications

The previous section shows that there is a big gap between usage statistics of contracts and contract codes. While the percentage of tokens in all contract codes is quite high (one quarter), the portion of actual smart contracts implementing a token is drastically lower (one percent). The reason for that is that some contract codes are deployed hundreds of thousands of times on the blockchain. In this section, we have a closer look at those contracts.

```sql
SELECT s.*, min(t.blockNumber)
FROM (
       SELECT hash, occurrences
       FROM eth.contractCode
       ORDER BY occurrences DESC
       LIMIT 10
       ) AS s
LEFT OUTER JOIN contract c
ON s.hash = c.contractHash
LEFT OUTER JOIN contractTransaction t
ON c.transactionHash = t.hash
GROUP BY s.hash, s.occurrences
ORDER BY s.occurrences DESC;
```

Figure 6.4: SQL query to get the first address of the most used contract codes

#	**First address**	**First block number**	**Occurrences**
1	`0xdf8c6179eccd33e69e0f049c799c1dd1cdf0e4e9`	4,150,297	1,577,003
2	`0x373b9ffa3168bfac15ab8b626e2fc98ad5d275a8`	4,978,201	633,871
3	`0x7622baab37c769a1f50079dfd1aa7b77a2c8976f`	4,995,470	540,103
4	-	-	540,102
5	`0x081d01a3ed60276d75da093033432fea328bf268`	3,750,675	391,512
6	`0x7538f0ad127a8a4ed0156f671a219e3447b84627`	4,545,729	311,166
7	`0x5aa66081b8ff73aac7d905f32d434a680635fd43`	4,269,936	306,611
8	-	-	176,910
9	`0xf2a96221c9393f0fd5b8a98e37c216bc3909847a`	3,712,109	142,581
10	`0xdbe25a28bf45bbacaf0fec554c3fa07053466a00`	4,353,450	125,327

Table 6.5: First address of the most used contract codes

Figure 6.4 shows the non-trivial SQL query used to obtain the data in table 6.5. The table lists for the top ten most used contract codes the first deployed address that used the contract code, its block number, and how many times the code is deployed on the blockchain. Note that two contract codes are missing the first two attributes. These contract codes are exclusively created by other contracts instead of externally owned accounts.

In order to identify the purpose of these contract codes, we use multiple sources from the Internet. For the most used contract code, the verified source code is publicly available on Etherscan[3]. There is a comment in the code saying that it is from the *Bittrex* cryptocurrency exchange, which, after some more investigation, seems very credible. Also, the number 9 most used contract code seems to be a contract created by Bittrex as well. All except for one contracts with that code are created by the same address and the

[3]`https://etherscan.io/address/0xdf8c6179eccd33e69e0f049c799c1dd1cdf0e4e9#code`

block explorer website *Bloxy* associates Bittrex with the address[4]. On Bloxy, there is also information about an address that created 99.96% of all contracts with the number 2 most used contract code[5]. These belong to an exchange platform, too, in this case *Upbit*. By the same reasoning, the number 6 most used contract code can be attributed to the *Ethereum Name Service* (ENS)[6]. And finally, the fifth contract code in our list is linked to the *Poloniex* exchange, as a Google search for its almost sole creator address reveals[7].

In summary, we are able to determine the purpose of half of the top ten most used contract codes. Four of these are cryptocurrency exchanges, which alone make up 39.3% of all smart contracts deployed on the blockchain. It is safe to say that this is the biggest area of application for Ethereum smart contracts in terms of amount of deployed contracts because many more cryptocurrency exchanges are known besides the three mentioned.

6.4 Metadata Analysis

This section, along with the next two sections, deals with research question 3: *"What does metadata tell us about the network?"* For our metadata analysis, we extract hard-coded addresses in smart contracts and apply the *PageRank* algorithm [56] to them. Afterwards, we observe which function signatures are used most often in contracts. Finally, we compare balances of smart contracts that were all created by the same contract.

6.4.1 Hard-coded Addresses

When a smart contract contains a hard-coded address, it appears as a `PUSH20 ...` operation in the EVM bytecode since addresses in Ethereum are 20 bytes long. Conversely, not every `PUSH20` operation necessarily pushes an address to the stack, but it is a strong indicator. Two special values of `PUSH20` operations that are not addresses are `0xff..ff` and `0x00..00`. For our analysis, we filter these values out. The script `extract_hard_coded_addresses.py` iterates over all contract codes, finds all instances of `PUSH20` instructions, and prints a ranking of the most used addresses in smart contracts. Table 6.6 lists the top five of that ranking.

Upon closer investigation, we discover that the number 1 most used address is used almost exclusively in the number 3 most used contract code and the number 2 most used address practically only appears in the number 4 most used contract code (see section 6.3.2). Behind both of these addresses is a contract with no transactions that does not give any information about why it is referenced so often. Only the fact that the creator of both of these contracts is identical (address `0x1Ff21eCa1c3ba96ed53783aB9C92FfbF77862584`) hints that the two most referenced addresses and therefore the number 3 and 4 most used contract codes are linked together. Metadata analysis helps to unearth such relationships.

[4]`https://bloxy.info/address/0xedce883162179d4ed5eb9bb2e7dccf494d75b3a0`

[5]`https://bloxy.info/address/0x4f01001cf69785d4c37f03fd87398849411ccbba`

[6]`https://bloxy.info/address/0x6090a6e47849629b7245dfa1ca21d94cd15878ef`

[7]`https://www.reddit.com/r/ethereum/comments/6ee36x/is_the_network_hanging_in_there_okay/di9mz9c/`

#	Address	Occurrences	PageRank
1	`0xc3b2ae46792547a96b9f84405e36d0e07edcd05c`	544,847	0.220 (1.)
2	`0x072461a5e18f444b1cf2e8dde6dfb1af39197316`	540,105	0.218 (2.)
3	`0xaf1931c20ee0c11bea17a41bfbbad299b2763bc0`	125,327	0.051 (3.)
4	`0xc8b55c7ad00fb9b933b0a016c6cebceea0293bb9`	112,905	0.015 (9.)
5	`0x208123a89e93fbbbeb19513315fee23698520029`	89,136	0.036 (4.)

Table 6.6: Most used addresses in smart contracts

Graph Analytics

The PageRank algorithm was originally developed by Page et al. to objectively assess the importance of web pages [56]. However, it can be applied to any graph to get nodes that are central or important. In `page_rank_addresses.py`, we build a directed graph from all addresses that appear in the bytecodes of smart contracts. The graph contains an edge from address A to address B, if and only if the code at address A contains a reference to B. We apply the PageRank algorithm to this resulting graph. In the last column of table 6.6, we note the PageRank score (which is between 0 and 1) and the absolute ranking among all nodes for the five most used contracts. Evidently, the contracts with the most occurrences also have the highest PageRank scores.

The reason why the results of the PageRank algorithm is not very meaningful is the structure of the graph. First of all, every node that represents the addresses of an externally owned account is a sink. Additionally, the graph is very sparse, containing 10,675 weakly connected components for the 2,103,225 total nodes. There are four non-trivial strongly connected components of the graph that contain two nodes each, meaning that these contracts reference each other. One example are the addresses `0xd40405fd33e1e5b15242fe8354a82f0663eb9540` and `0xa27c1a216db8027e5b4a77c2f4fca0ed88db166d`. Upon the creation of the first contract, its creator must have anticipated the creation of the second one. However, none of the four contract-pairs contains any Ether or is of special interest. When removing all cyclic edges of strongly connected components, the longest path in the graph contains nine nodes.

6.4.2 Function Hashes

Besides hard-coded addresses, smart contract bytecodes also reveal function signatures. Table 6.7 shows the top ten most used function signatures of all contracts that were determined using the script `extract_function_hashes.py`. Finding function signatures works just like finding hard-coded addresses, but this time we observe `PUSH4 ...` operations. We filter out the values `0xffffffff` and `0x01000000` because they do not correspond to function signatures. When outputting the ranking of the most used signatures, we perform a lookup in our `functions` table to get the function name, if available. For three signatures in the top ten, no function name is known. These three signatures are used in the fifth most used contract code 391,512 times. As seen in section 6.3.2, these contracts belong to the Poloniex cryptocurrency exchange. We can assume that these functions provide some functionality specific for the exchange.

#	Function signature	Function name	Occurrences
1	`0x3c18d318`	`sweeperOf(address)`	2,478,244
2	`0x6ea056a9`	`sweep(address,uint256)`	2,469,444
3	`0xa9059cbb`	`transfer(address,uint256)`	1,881,524
4	`0xc0ee0b8a`	`tokenFallback(address,uint256,bytes)`	1,803,067
5	`0x70a08231`	`balanceOf(address)`	1,238,426
6	`0x8da5cb5b`	`owner()`	1,021,211
7	`0xe5225381`	`collect()`	446,953
8	`0xb76ea962`	-	395,259
9	`0x82c90ac0`	-	391,522
10	`0x66117276`	-	391,516

Table 6.7: Most used function signatures in smart contracts

The two most used functions `sweeperOf(address)` and `sweep(address,uint256)` are functions that are used by smart contracts of many cryptocurrency exchanges to transfer Ether and tokens. Both Bittrex and Upbit have these functions exposed in their contracts codes. `transfer(address,uint256)` and `balanceOf(address)` are part of the ERC20 standard and are therefore used by all ERC20 token implementations [54]. The other functions are auxiliary functions. `tokenFallback(address,uint256,bytes)` is defined by the ERC223 standard [57] and is used to handle token transfers from ERC20 token contracts. `owner()` returns the creator of the smart contract and `collect()` is used by the initiators of crowd sales to collect their money if the goal is reached.

In all, there are 204,879 different function signatures in the Ethereum network and 27,589,209 total functions. On average, a smart contract contains 3.95 functions.

6.4.3 Balances of Created Contracts

For our last analysis in this section, we look at accounts (contracts and externally owned accounts) that created the most smart contracts. Additionally, we are interested in the balances of these accounts. The script `get_balances_of_created_contracts.py` gathers data from both the `contract` and the `contractCreatedContract` database table and leverages the `eth.getBalance()` call to determine the balance of the creator and to calculate the sum of the balances of the created contracts. The results in table 6.8 correlate with the most used contract codes from section 6.3.2 because often a contract code that occurs many times on the blockchain is created by the same account. For example, the account that created the most smart contracts only created contracts with the contract code that was used most often. As mentioned in section 6.3.2, this account belongs to the Bittrex cryptocurrency exchange. The creator addresses of the numerous Upbit and Poloniex smart contracts are also represented in table 6.8 at position 3 and 4, respectively. Also, the main contract of the Ethereum Name Service (ENS) is here in sixth position. There is a high amount of money in its child contracts because the service is used by many different users for registering domain names for addresses. After the initial request for a domain name by a user, the main contract creates a new deed contract where users can place bids for that domain name. The sum of the balances of all child contracts of the ENS registrar is the amount of Ether that is currently in auctions for domain names.

#	**Creator address**	**Creator balance**	**Contracts created**	**Sum of child balances**
1	`0xa3c1e324ca1ce40db73ed6026c4a177f099b5770`	0 ETH	1,576,972	615.93 ETH
2	`0x71d271f8b14adef568f8f28f1587ce7271ac4ca5`	0 ETH	743,646	865.25 ETH
3	`0x4f01001cf69785d4c37f03fd87398849411ccbba`	0 ETH	633,629	0 ETH
4	`0xb42b20ddbeabdc2a288be7ff847ff94fb48d2579`	3.36 ETH	391,521	26.27 ETH
5	`0x279b045989bd4cd60ee4a53d2a1c0621a4b4623f`	0 ETH	333,422	96.57 ETH
6	`0x6090a6e47849629b7245dfa1ca21d94cd15878ef`	0 ETH	311,160	177,038.34 ETH
7	`0x17bc58b788808dab201a9a90817ff3c168bf3d61`	22,192.28 ETH	306,610	124.63 ETH

Table 6.8: Balances of accounts that created the most child contracts

One application of this methodology is to gain insights in how big cryptocurrency exchanges are. We can assume that these exchanges are creating hundreds of thousands of smart contracts in order to have one contract per user. The advantage of this approach over having one single large contract with all assets of all users is that there is no single point-of-failure and it facilitates user token management. The amount of contracts gives us an estimation of the number of users of a cryptocurrency exchange. Also, by summing up the balances of the contracts, we can easily get an idea of how much money users have deposited at the exchanges. For exchanges that manage all their users in one single large contract, the number of users cannot be estimated so easily.

6.5 Approximating Compiler Versions

A very important piece of metadata information is which Solidity compiler version was used to create the EVM bytecode. Of course, it is also possible to skip the compiler and directly write EVM bytecode or to use another language that compiles into EVM code, like Vyper[8] or LLL[9]. However, as Solidity is the most widely used language to write smart contracts, we only consider that. In the compiled bytecode itself, there is no dedicated field for the compiler version. Therefore, we use two different heuristics to derive a range of compiler versions that could have been used. The script `determine_possible_compiler_versions.py` calculates these heuristics for every contract code and updates the `minCompilerVersion` and `maxCompilerVersion` columns in the database.

In section 6.6 we are able to further refine the compiler version range by leveraging library versions. We evaluate our compiler version approximation in section 7.2.1.

[8]`https://github.com/ethereum/vyper`
[9]`https://lll-docs.readthedocs.io/en/latest/lll_introduction.html`

6.5.1 Contract Creation Date

The first heuristic we use for compiler version estimation is the contract creation date. The reasoning behind this is straightforward: a smart contract cannot be compiled with a compiler version that was not released yet at the time of the smart contract creation. Therefore, for every contract code we get the timestamp of the oldest user-created contract with that code from our database. The timestamps are derived directly from the block header of the block, which the contract-creation transaction was part of. In case a contract code is only used by contract-created contracts, we use the creation date of the oldest parent because the child contract uses the same compiler version as the parent.

In the `compiler` table, we manually inserted the release dates of all Solidity compiler versions from `0.1.1` (released in August 2015) to `0.5.2` (released in December 2018). This information is publicly available on the GitHub releases page of the Solidity compiler [43]. With that, we can create a range for each contract code, starting from the earliest compiler version and ending with the compiler version released just *after* the first contract with that contract code was released. We intentionally set the upper boundary of the range one version too high, because a nightly version could have been used to compile the contract. Nightlies are pre-releases for development purposes that may already contain some of the features for the next release. Usually, they are released every night, hence the name. As soon as a new stable version is released, a new nightly for the next stable compiler version is released as well. In all, this heuristic gives an upper bound for the compiler version.

6.5.2 Header Analysis

For our second heuristic we look at headers of smart contracts, i.e. the first few bytes that their bytecode starts with. With `find_bytecode_prefixes.py`, we determine the frequency of all one- to ten-byte prefixes and create a ranking. The flow chart in figure 6.5 shows the most relevant prefixes for our analysis. About 91.8% of all smart contracts start with `0x6060604052`. These are two `PUSH1` operations and an `MSTORE` operation that are used to initialize the memory pointer. The Solidity compiler always inserts these instructions at the very beginning of the generated bytecode. However, with version 0.4.22 of the Solidity compiler the initialization slightly changed and the first operation is not `PUSH1 0x60` anymore, but `PUSH1 0x80`[10]. Therefore, if a contract starts with `0x60806040`, it is very likely that this contract was compiled with a compiler version greater than or equal to 0.4.22.

Now we take a closer look at contracts starting with `0x6060604052`. For 99.7% of all these contracts, the next byte in the bytecode is one of either `0x5b`, `0x63`, `0x36`, or `0x60`. One header that we are able to associate with a compiler version range with high certainty is `0x60606040526004`. Contracts starting with this prefix are very likely to be compiled

[10]The commit of this change can be found here: `https://github.com/ethereum/solidity/commit/0cbe55005de79b0f7c5c770d50c3eb87df019789#diff-6ec42630506b86abb9d7386116dd0e5bR53`

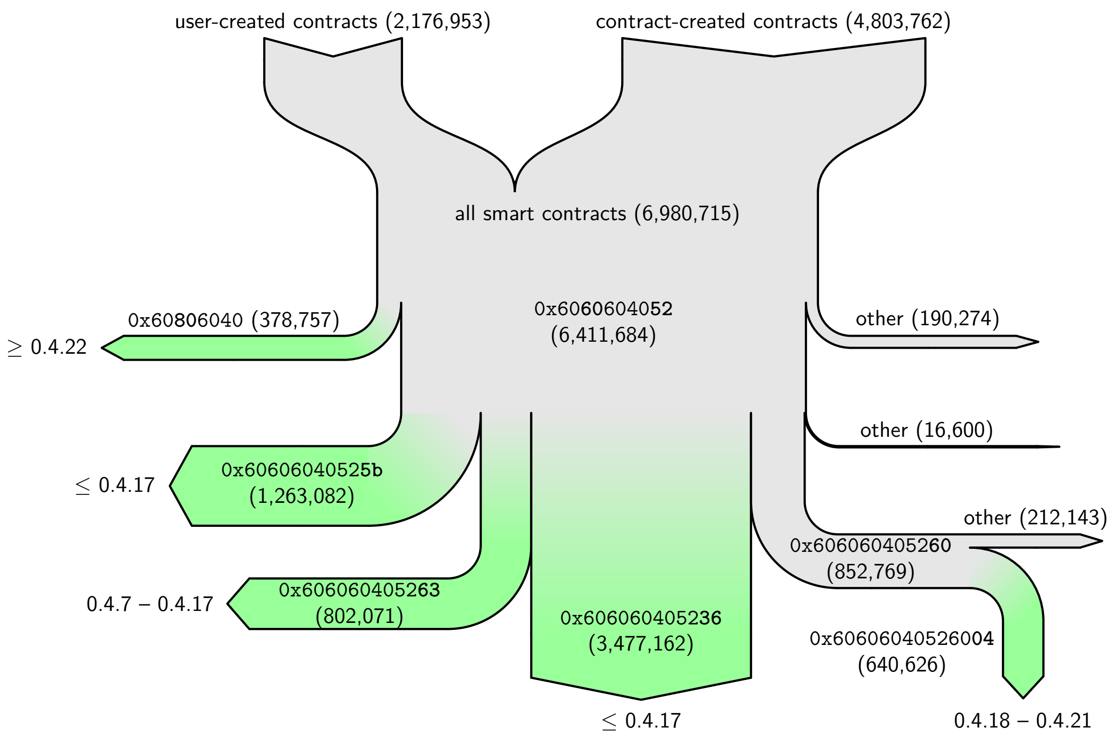

Figure 6.5: Most used prefixes of smart contract bytecodes

with a version between 0.4.18 and 0.4.21 (inclusive)[11]. For other contracts starting with `0x606060405260` (without the additional `04`), no definitive statement about the compiler version can be made.

From the previous findings, we conclude that contracts starting with `0x60606040525b`, `0x606060405236`, or `0x606060405263` are very likely to be compiled with version 0.4.17 or lower. After compiling many smart contracts and inspecting the resulting bytecode, we can narrow down one header even more. Smart contracts starting with `0x606060405263` are compiled with a version greater than or equal to 0.4.7 where compiler optimizations are turned on. The five headers that we identified are colored green in figure 6.5.

6.6 Library Usage

Another relevant piece of metadata information is whether a smart contract is using libraries and if yes, which library version is used. We only focus on the *SafeMath* library from OpenZeppelin [46] here because it is by far the most popular Solidity library. However, these techniques can be applied to other libraries, too.

As Solidity does not protect arithmetic operations from over- or underflowing, the

[11]This is the relevant commit in the Solidity compiler repository: `https://github.com/ethereum/solidity/commit/a3db1fc1976e1b2e67aedecb771c288b6dca6b1c#diff-442991761f888e625e1df9f83e8ff779R256`

programmer has to check manually for these errors. SafeMath provides functions that replace ordinary arithmetic operations and always check for over- and underflows at every operation. The first version of the SafeMath library was released in November 2016 and it only contained the replacement functions `mul`, `add`, and `sub`. The last version that we consider is 2.1.1 released in January 2019, which additionally has a safe version of the `div` and a `mod` operations. Some SafeMath versions also contained `min` and `max` functions that were later refactored into a separate *Math* library. However, because these two functions are so fundamental and sometimes appear in smart contracts that do not include the library, but simply implement these exact functions themselves, we do not consider them for our analysis. All SafeMath functions are `internal`, meaning that their code is copied into the bytecode of every contract that is using the library.

Our approach to detect SafeMath usage in Ethereum smart contracts consists of three steps. First, we compile all versions of the library with all compatible compiler versions. Then, we extract the bytecodes of each individual library function from the compiled libraries and store them in the database. And finally, we search through all contract codes on the blockchain to see if they contain one or more of the extracted library functions.

6.6.1 Compiling All SafeMath Libraries

We downloaded all SafeMath versions from OpenZeppelin's official GitHub releases page [46]. However, when compiling only the library together with an empty contract, the compiler does not put the bytecode of the internal library functions inside the contract bytecode. In order for that to happen, these functions must be called at least once by the contract. This mechanism avoids dead library code in the final contract and therefore reduces gas consumption. For that reason, we write a test contract for every SafeMath version that calls each internal library function once. Every call is done with a unique parameter that makes it easy to find the call in the bytecode later on, e.g. `a.mul(0x1111);`. The test contracts slightly vary for different library versions because some older versions do not contain the `mod` and `div` functions. Additionally, until version 1.0.4, the library was not implemented as a Solidity `library`, but as a regular `contract` and the functions were named differently. Because of these subtleties, it is not easily possible to create the test contracts automatically. Instead, they must be adjusted manually for every library version. However, for the remaining analysis of the resulting bytecodes, the minor differences in the library versions do not play a role.

The Node.js project in the `multi_compile_contracts` folder of our repository allows compiling a Solidity program with all compatible compiler versions. Using a Bash script, we execute this program for each of our SafeMath test contracts. Internally, the project uses the Emscripten compiled binaries of the Solidity compiler [58]. For every compiler version that is compatible with the `pragma` header of the contract, the program automatically compiles the contract both with and without optimizations turned on. The output bytecodes are then written to files in a dedicated directory and are later used by the library function extraction tool.

6.6.2 Extracting Library Functions

After compiling all SafeMath libraries with all compatible compiler versions, we need to extract the bytecode of the internal functions from the raw bytecode. Zhou et al. describe a sophisticated heuristic to identify internal functions in a general way [29]. We, however, implement a simpler heuristic in the script `extract_library_functions.py` by taking advantage of the fact that we call each internal function with a unique parameter in our test contracts. As a starting point for our search for an internal function, we take the `PUSH2` instruction that pushes our unique parameter for that function onto the stack. Of course, when picking the parameters, we make sure that they do not appear anywhere else in the bytecode. From the starting point, we search for the next `PUSH2` instruction in the bytecode because after the parameter is loaded, the jump to the internal function is prepared. The `PUSH2` instruction loads the address of the beginning of the internal function onto the stack and immediately afterwards a `JUMP` to that address follows. That marks the beginning of the internal function. Our experiments show that there are three possible sequences of instructions that can end the internal function: `JUMPI INVALID STOP` or `REVERT STOP` or a `JUMP` without no `PUSH2` instruction directly preceding it. After determining the exact beginning and end of an internal function, we have to sanitize the bytecode. That means that we remove all absolute jump destination addresses in the bytecode in order to make it comparable, no matter at which offset the internal function is located within the contract. Therefore, we replace the parameter of every `PUSH2` operation with the placeholder `xxxx`. Finally, we save the sanitized function bytecode in the `libraryFunction` database table, together with the compiler version, a flag whether compiler optimizations were turned on, the library version, and the function name.

From the 540 compiled binaries from the previous section, we are able to extract 82 distinct library function bytecodes.

6.6.3 Finding Library Occurrences

The final step of detecting SafeMath library usage is to match the extracted library functions with smart contract bytecodes, which we do with the script `find_library_occurrences.py`. We iterate over all contract codes and for every one of the five SafeMath library functions (`add`, `sub`, `mul`, `div`, `mod`), we check if we find any of the extracted function bytecodes in the contract code. As mentioned, a contract only contains the bytecodes of internal library functions that it actually calls at some point. Next, we create a list of compiler and library version combinations that are possible for this exact combination of functions. To do so, we request all versions from the `libraryFunction` table for each individual matched function bytecode. Our initial preliminary list of compiler and library version combinations consists of those combinations that are possible for *all* matched functions. We further refine this list two times. Combinations with compiler versions that are outside of the range determined by our previous compiler version determination (see section 6.5) are discarded. Moreover, similar to our compiler version approximation, we can set an upper boundary for the maximum library version that could have been used by looking at the release date of the library. A contract cannot have used a SafeMath library version that was released much later than the contract itself. Again, due to pre-releases of the library we also consider one library version that was released `after` the contract. Lastly, we update the `contractCode` database table with these upper and lower boundaries of both library and compiler versions. For every set of library function bytecodes there is only a limited number of library and compiler version combinations meaning that with this technique we can narrow down the estimated compiler version even more from our previous approach.

In total, we were able to detect 87,575 smart contracts with 48,304 distinct contract codes that use the SafeMath library. That is more than one quarter of all contract codes. This result resembles the outcome of our token usage statistics (see section 6.3.2). It shows that on the one hand there are a few contract codes with little functionality that are represented millions of times on the blockchain and on the other hand there are many contract codes often uniquely represented by a single instance of a smart contract that implement sophisticated behavior. Two indicators that a contract is probably doing something more complex and non-trivial are the usage of advanced libraries and the implementation of an ERC token standard.

7 Evaluation

We assess the quality of the results from the previous chapter next. After detailing our evaluation approach, we evaluate those of our findings that are not exact facts. Particularly, that encompasses the compiler and library version estimation.

7.1 Evaluation Approach

In order to be able to evaluate our version estimations, we need a data set that contains smart contract bytecode, the corresponding source code, and the compiler version that this code was compiled with. The source code is important to check whether the SafeMath library was indeed used or not. Verified contracts on Etherscan provide all of these properties.

7.1.1 Verified Contracts on Etherscan

During previous research, another member of our chair built a web crawler that downloads all verified contracts from the Etherscan website and saves them to a database, together with all publicly accessible attributes provided by the website like the compiler version, the verification date, and more. The data set contains 134,308 verified contracts. 54,179 of them are from the main network, which we are focusing on in our analysis. The rest of the verified contracts are from one of the Ethereum test networks.

We integrate the `verifiedContracts` table into our database setup. The script `match_verified_sourcecode.py` matches the entries of our `contract` and `contractCreatedContract` table with the verified contracts based on the address and the network ID. Then it sets the `verifiedSourceCodeID` field of all contract codes for which a verified contract was found. In total, we have 50,433 different contract codes with verified source codes making up 2,096,497 smart contracts. This number is a bit lower than the total number of contracts from the main network in the `verifiedContracts` table because the table contains contracts that were created after block 6,900,000 and because some verified contracts from Etherscan are using identical contract codes. Also, we disregard all verified contracts in the database table that do not have the `compiler` field set. Nevertheless, more than 50,000 verified contract codes is a good evaluation basis.

7.1.2 Limitations

Our evaluation approach comes with a few limitations that must be mentioned. First of all, some of our results cannot be evaluated well at all. For example, the detection of ERC20 and ERC721 tokens is not an estimation, but a fact based on whether a contract bytecode contains certain four-byte function identifiers. If we were to search for the real function names in the source code of verified contracts to check whether a contract implements a token, we would come to the same result. The same is true for our other general statistics about the network, like most used contract codes or transactions to future contracts.

Furthermore, there are limitations of the data set provided by Etherscan. Of the 54,179 verified contracts from the main network, only 691 or 1.28% are using a compiler version lower than 0.4.0. This number is disproportionate to all contracts on the blockchain with a version number lower than 0.4.0. Version 0.3.6 (which was the version *before* 0.4.0) was released on August 10, 2016 and until that point in time, there were 55,889 user-created smart contracts on the blockchain, which is 2.57% of all user-created contracts now. These are all contracts that definitely must have used a version lower than 0.4.0, even considering the nightly releases of 0.4.0. Moreover, it is safe to assume that many programmers still used older compiler versions even after the release of 0.4.0. So in order for the number of verified contracts with compiler version lower than 0.4.0 to be proportionate to the number of all contracts that number would need to be at least twice as high as it is.

Research has shown that data from Etherscan cannot be trusted unconditionally. There are several verified contracts for which the compiler version from Etherscan does not fit to the compiler version in the source code. For example, Etherscan claims that the contract at address `0xbb13b668fa4a9cf89c601997355d759b0f7799a9` was compiled with compiler version *v0.4.14+commit.c2215d46*, but in the source code the very first line reads `pragma solidity ^0.4.18;` meaning that this code should be compiled with at least version 0.4.18. In the `verifiedContracts` table, there are fourteen more examples from all networks of such incompatible compiler versions. We manually checked that these are indeed mistakes by Etherscan, and not by the data crawler. It is unclear how these contracts were verified because compiling source code with an incompatible compiler version always aborts with an error. Even though these contracts make up merely 0.01% of all verified contracts, they lower the trust in the data provided by Etherscan. If such mistakes happen, there is a chance that there are other mistakes as well that are not as easily detectable. Nevertheless, since this is the biggest data set of smart contract source codes annotated with compiler versions we still use it for our evaluation.

Unfortunately, Etherscan recently made a change to their website showing only the last 1000 verified contracts as a list instead of all of them. That makes it harder to crawl all verified contracts because it is not possible anymore to go through all the pages of the list and scrape the contracts one after another. Instead, one would have to make the crawler access the web page of one smart contract for every contract code in our database. If that contract, or a “similar” contract as Etherscan calls it, is verified, the source code can be obtained that way.

7.2 Evaluation Results

After describing our evaluation approach and its limitations, we now present the evaluation results of the compiler and SafeMath version estimation. All results are obtained with the scripts `verify_results.py` and `calculate_significance_of_version_numbers.py`.

7.2.1 Compiler Version Estimation

Of the 50,433 contract codes in our evaluation set, our approach determines a correct minimum and maximum compiler version for 50,061 or 99.26% of them. In terms of total contracts, that is 2,059,098 of 2,096,497, or 98.22%. More than two thirds of the contract codes, for which the estimation of the compiler version is incorrect, are using a nightly compiler version. This is the main reason why our heuristics fail, even though we are considering nightly versions in our compiler version estimation. However, we do not compile all SafeMath libraries with nightly versions because there would have been too many compiled outputs. Until compiler version 0.5.2, there are 46 stable and 621 nightly releases. That is why the estimation for contracts that are using a nightly version *and* are using the SafeMath library is sometimes slightly wrong.

One other example of where our heuristics determine a compiler version that differs from the one on Etherscan is for the contract at address `0xbb13b668fa4a9cf89c601997355d759b0f7799a9`. As mentioned in section 7.1.2, the "verified" version on Etherscan for that contract (0.4.14) does not fit to the version in the source code (0.4.18 or above). For this contract, our approach determines the compiler version to be between 0.4.18 and 0.4.20, which fits better to the source code version than the verified version on Etherscan.

Compiler Version Distances

These results alone do not suffice to judge the quality of our estimation. A hypothetical compiler estimation heuristic that sets the minimum compiler version to 0.1.1 and the maximum version to 0.5.2 for all contracts would be correct for every single contract but the results would be meaningless. Therefore, we also evaluate the size of the range of our estimation, i.e. how many version numbers the minimum estimated version and the maximum estimated version are apart from each other. In the optimal case, where the upper and lower boundaries are identical, this number would be zero, whereas the maximum distance is 45. This time, as an evaluation basis we do not take the 50,433 contract codes that have verified source code, but all 190,226 contract codes. The average compiler version distance of these contract codes is 11.45 with a median of 3. Considering all smart contracts, the average distance is 24.82 and the median is 33. However, since a few contract codes are represented hundreds of thousands of times on the blockchain, the last two numbers do not have as much meaning as the first two. For example, there are 2,159,645 smart contracts with distance 33, but only 1,128 contract codes with that distance. That is equivalent to 30.9% of all smart contracts, but only 0.6% of all contract codes, which is why the average and median are so high when looking at all smart contracts.

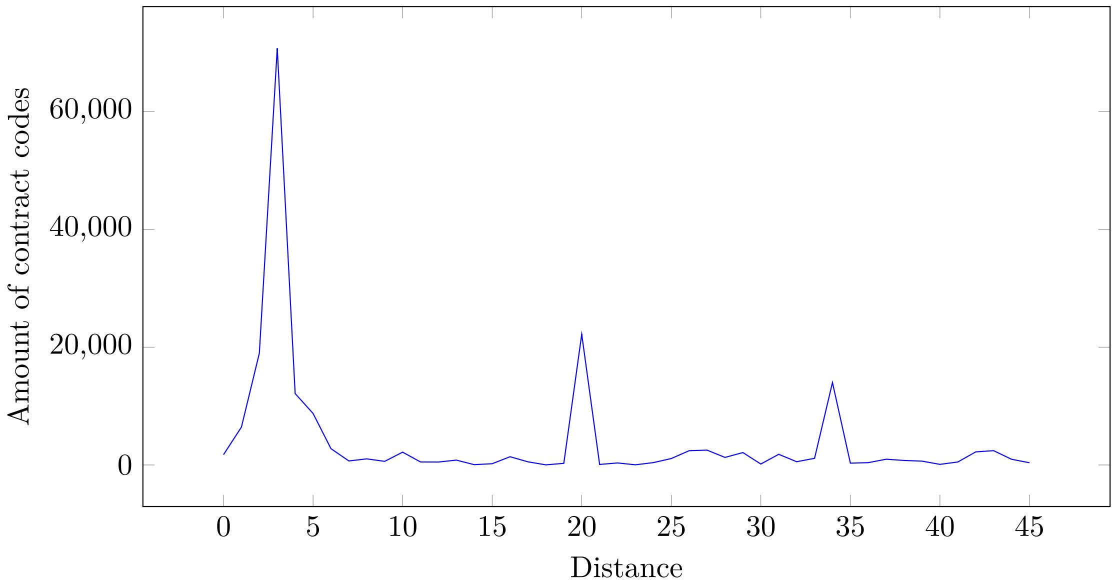

Figure 7.1: Distances of compiler version estimations for contract codes

Figure 7.1 shows the amount of contract codes for every possible compiler version distance. There are three distinct peaks at distances 3, 20, and 34. These originate directly from our methodology. In section 6.5.2, we identified the header `0x60606040526004` to be corresponding to compiler versions between 0.4.18 and 0.4.21. These two boundaries have exactly distance three and are responsible for a large part of the peak at that distance. Another large fraction of that peak stems from the header `0x60806040`, for which we set the lower compiler version boundary to 0.4.22. The time between releases 0.4.24 and 0.4.25 was with almost four months unusually long. Normally, a new version of the Solidity compiler is released every one or two months. For all contracts in this four-month period, our heuristic sets the upper boundary to 0.4.25 (due to our adjustment for nightly releases). Together with the lower boundary, we get a total compiler version distance of 3 for these contracts.

The peak at distance 34 can be explained similarly. For the headers `0x606060405236` and `0x60606040525b`, we set the upper boundary to version 0.4.17, which is the 35th compiler release, making the distance to the first compiler version 34. Also, there is a very small peak at distance 10, which is due to the header `0x606060405263`, which we determined for compiler versions between 0.4.7 and 0.4.17. This peak is so small because the header only occurs in contracts that were compiled with compiler optimizations.

At distance 20, there is a peak with a different explanation. Between block 2,370,025 and 2,423,365 three spam accounts[1] put 23,641 smart contracts onto the blockchain, most

[1]addresses `0x3898D7580aa5B8aD8a56fCd7f7Af690e97112419`, `0x40525aC2Fe3bEFE27A4e73757178d4aCCfEF71dA`, and

of which have different contract codes. Our heuristic from section 6.5.1, which uses the contract creation date to set an upper bound for the compiler version, determined compiler version 0.4.3 for all of them as maximum version. That yields a distance of 20 and explains the high peak.

For 1,751 contract codes our approach sets the upper and lower boundary to the same compiler version, i.e. they have distance zero. 642 of them can be mapped to a verified source code for evaluation. Of those, 613 (95.5%) are estimating the compiler version correctly.

False Positives in Header Analysis

Now we have a closer look at our bytecode header analysis and evaluate each of our header heuristics from section 6.5.2 separately. Table 7.1 lists for every one of our five identified headers how well they estimate the specified compiler version. There are four possible categories:

- **True positives** – These are contract codes having the specified header and being compiled with the specified version.
- **True negatives** – They do not have the specified header and have a different version.
- **False positives** – Contract codes in this category have the header but they are compiled with a different compiler version.
- **False negatives** – These contract codes are compiled with the specified version, but do not have the header.

Ideally, we would like to have only true positives and true negatives because then the header estimation would never make mistakes. However, this is not realistic and in our case, false negatives are not as big of a problem either. A high number of false negatives indicates that there are contract codes with that compiler version that do not have the specified header. Consequently, there must be other headers for that compiler version. The amount of false negatives for the last three rows in table 7.1 are quite high because all three are covering the compiler versions below 0.4.17. When evaluating these three headers combined (for version $\leq$ 0.4.17), the number of false negatives drops to just 464, or less than 1% of all contract codes in the evaluation set. In our estimation, these false negatives mean that the version estimation is more conservative for them and the distance between the minimum and maximum compiler version is higher than for true positives or true negatives.

More severe than false negatives are false positives. In these cases, our algorithm assigns a wrong compiler version range to contracts. The reason for most of the errors in this category are once again nightly versions. More than three quarters of the 288 false positive results are due to nightly versions. For two of our headers there are no false positives at all

0x1FA0e1DFa88b371fcEdf6225b3d8ad4e3baCeF0E

Header	**Version**	**True positives**	**True negatives**	**False positives**	**False negatives**
`0x60806040`	$\geq$ 0.4.22	22,104	28,028	0	301
`0x60606040526004`	0.4.18 – 0.4.21	19,100	30,930	264	139
`0x606060405263`	0.4.7 – 0.4.17	329	42,550	2	7,552
`0x606060405236`	$\leq$ 0.4.17	7,960	41,622	22	829
`0x60606040525b`	$\leq$ 0.4.17	36	41,644	0	8,753

Table 7.1: Bytecode header heuristics evaluation

and for the other three headers the total number of false positives is so low in comparison to the total number of contract codes that these wrongly identified contract codes are negligible.

7.2.2 SafeMath Library Usage and Version Estimation

The last result we evaluate is the usage detection of the SafeMath library and its version estimation. For evaluation, we extract information whether a verified contract uses the SafeMath library directly from the source code. If a source code contains a library or contract named `SafeMath` or `SafeMathLib`, or if there is a function named `add`, `mul`, `div`, `sub`, `Add`, `Mul`, `Div`, `Sub`, `safeAdd`, `safeMul`, `safeDiv` or `safeSub`, we state that the contract uses the SafeMath library. Of course, this is only a heuristic, but arguably a much better heuristic than the one described in section 6.6 with which we extract this information from the bytecode alone. This is why we can use this heuristic to evaluate the other one from the previous chapter. The evaluation set again consists of our 50,433 verified contract codes. Table 7.2 shows how SafeMath usage detection using the source code correlates with usage detection by only looking at contract bytecodes. In 27 cases, our approach detected a SafeMath library from the bytecode, but not from the source code. All of these contract codes indeed use safe math functionality, but the code contains unusual function names, like `safeMultiply` or `divide`. If we expanded our search keywords to more function names, we would risk including functions that are associated with the SafeMath library.

	Source code uses SafeMath	**Does not use SafeMath**
Detected SafeMath in bytecode	21,375 true positives	27 false positives
Did not detect SafeMath	8,978 false negatives	20,053 true negatives

Table 7.2: SafeMath usage evaluation

The number of false negatives is with 17.8% much higher. In these cases our bytecode heuristic fails, which can have multiple reasons. Some contracts are including the SafeMath library in their code without using it. As mentioned in section 6.6.1, if an internal function from a library is not called anywhere in the contract, the bytecode of that function is not included in the final bytecode. In that case, our bytecode heuristic from the previous chapter is not able to detect anything, while the source code heuristic detects the existence of the library. One example of a contract not using the included library can be found at address `0x821cc0b6817ea2b23c87b591d72328a821edf694`. Another reason for false negatives is that we do not have every possible bytecode of compiled SafeMath functions in our database. Under some circumstances, the compiler adds one or more additional instructions to the bytecode of SafeMath functions that make our detection fail. It is difficult to say how many more possible SafeMath function bytecodes there are. To determine such bytecodes is left for future research.

SafeMath Version Distances

We do not attempt to evaluate whether the detected range of SafeMath versions of our approach is correct because it is not possible to read the true SafeMath version number from verified contracts on Etherscan. We only evaluate the distances of our version estimation, similar to the compiler version distance evaluation from before. Continuing, we now take the 48,304 total contract codes as an evaluation basis, for which we were able to set a minimum and maximum SafeMath version. These contract codes make up 87,575 contracts. The average distance of contract codes is 4.96 with a median of 4. When looking at all contracts, the average is 4.66 and the median is 4 as well.

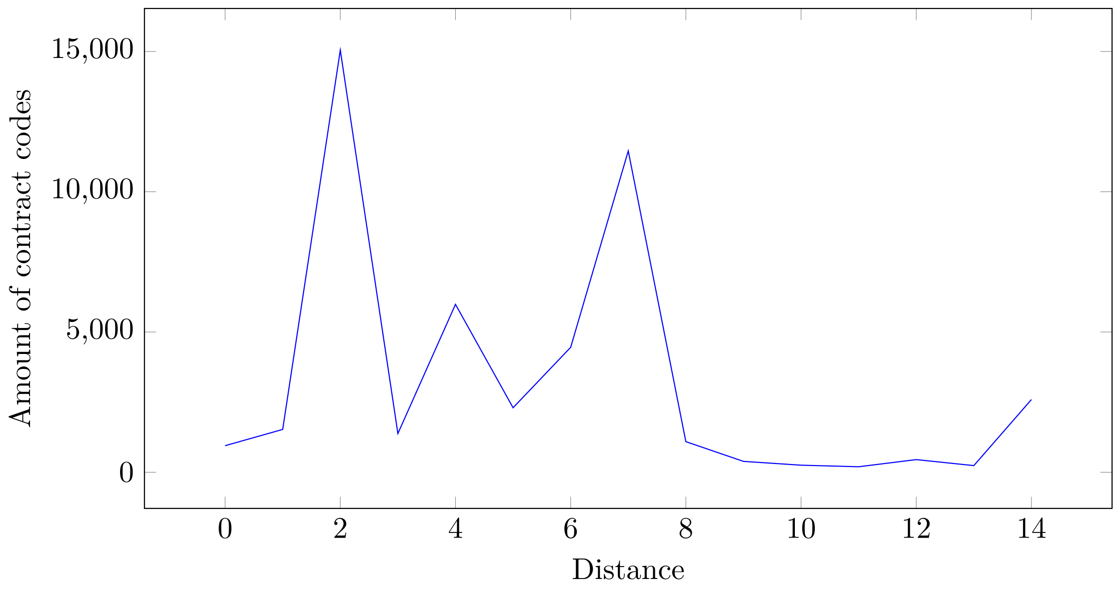

Figure 7.2: Distances of SafeMath library version estimations for contract codes

In figure 7.2 we plot the amount of contract codes for every library version distance. There are 31 SafeMath versions, including release candidate (RC) versions, but our approach manages to keep all version ranges at distance 14 or below. All contract codes with distance 14 have version 1.1.0 as minimum version and 1.12.0 as maximum version. These were exactly the two big version number jumps of the library (from 1.0.7 to 1.1.0 and from 1.12.0 to 2.0.0 RC1), indicating substantial changes in the code. From version 1.0.0 until version 1.0.7 SafeMath implemented its own internal `assert()` function that is called by the `add()`, `mul()`, and `sub()` function for over- and underflow checks. Starting from the following version 1.1.0 the library uses the `assert()` function provided directly by Solidity. That results in different bytecode of each math function for the two versions. Then, with version 2.0.0 RC1, the developers of the library switched from the `assert()` function to the built-in `require()` function[2]. That function compiles to slightly different bytecode again, making all versions greater than or equal to 2.0.0 RC1 unambiguously identifiable. Additionally, SafeMath version 2.0.0 RC1 introduced the `mod()` function. Existence of the bytecode of that function is a further indicator for version 2.0.0 RC1 or above. By leveraging these changes in the SafeMath code our approach is capable of determining a SafeMath version range of at most 14 for every contract code.

Figure 7.2 contains three other distinct peaks that we explain briefly. The range of 86% of the contract codes with library version distance 2 is 1.1.0 – 1.3.0. That is because version 1.4.0 introduced a modified `mul()` function, whose bytecode differs from its predecessor. Also, SafeMath version 1.4.0 requires Solidity compiler version 0.4.18 or higher, while SafeMath 1.3.0 has set 0.4.11 as minimum compiler version. Next, for about half of the contract codes with distance 4 we are estimating compiler versions 1.4.0 – 1.8.0. At version 1.9.0, there were clear changes to the `add()`, `mul()` and `div()` functions. Together with the changes from version 1.4.0, this peak is explained. Finally, 99% of all contract codes with a library distance of 7 are in the range 1.1.0 – 1.8.0. Again, this is due to the big code changes at versions 1.1.0 and 1.9.0.

To summarize, our SafeMath version estimation detects changes in the code of the SafeMath library. The biggest changes were at versions 1.1.0, 1.4.0, 1.9.0, and 2.0.0 RC1.

[2]The discussion can be found here: `https://github.com/OpenZeppelin/openzeppelin-solidity/issues/1120`

8 Conclusion and Future Work

In this chapter, we summarize the results of this thesis and give an outlook of possible future research. Especially the conceptual model we developed can be used as a basis for other theses and papers. Further analyses can be conducted with our database.

8.1 Conclusion

In this thesis, we extensively analyzed the Ethereum platform. To begin with, we defined all necessary cryptographic basics, explained key concepts regarding blockchain and other distributed ledger technologies, provided an overview of Ethereum clients, and defined the term conceptual model. Afterwards, we gave an overview of the state-of-the-art research about Ethereum by summarizing approximately 20 scientific papers and other publications from this field and from related areas. Next, we answered research question 1 by developing a conceptual model of the Ethereum system. After giving an overview of some fundamental concepts of Ethereum, we used UML class diagrams to structure the system into the four parts “Source”, “EVM”, “Storage”, and “Ledger”. We concluded the theoretical part of this thesis by showing correlations between the four parts.

Afterwards, we began with data acquisition for our practical analyses. With answering research question 2, we showed what data is available for analysis and evaluated several approaches of data extraction and storage. In the end, we used a relational database to store smart contracts separated from contract codes. In chapter 6, we first presented some general statistics about our data set encompassing the first 6,900,000 blocks. Then, we addressed research questions 3, 4, and 5 with a multitude of analyses. We gave examples of three categories of anomalies: front-running, self-destructing constructors, and transactions to future contracts. Following, we determined real-world use-cases of smart contracts on the Ethereum blockchain by measuring ERC standard usage and by manually inspecting some of the most used contract codes. For our metadata analysis, we began with extracting hard-coded addresses from contract bytecode and applying the PageRank algorithm. In addition, we looked at most used function signatures and accounts that created the most smart contracts and their balances.

Our main contribution is the approximation of compiler and library versions. By using heuristics based on the contract creation date and a header analysis, we were able to set very accurate ranges of compiler versions for arbitrary contract codes. We accurately detected usage of the SafeMath library by compiling all library source codes with all compatible compiler versions, extracting the bytecodes of the functions, and searching for them in all contract codes. Our evaluation with verified source codes from Etherscan attested that our compiler version estimation ranges are correct for 99% of all contract codes. The median distance of the minimum and maximum estimated version for contract codes is 3. For SafeMath usage detection, we have a success rate of 82% with a median distance of 4.

8.2 Future Work

There are multiple downsides of our approach of obtaining contract-created contracts. Most importantly, we do not know the block number of when such a contract was created and we do not retrieve all contracts. These drawbacks can be solved by using Parity traces. Parity – in contract to Geth – is capable of tracing message calls induced by a transaction. That means that if a transaction does not create a smart contract directly, but causes a contract to invoke the `CREATE` opcode, Parity can trace that message call. By replaying all transactions, it is possible to collect the complete list of contract-created contracts, including their creation date. That data would then for example allow making a statement about the distribution of contract-created ERC20 tokens over time.

Another aspect that has room for improvement is data collection for our compiler and SafeMath version estimation. It might be possible to reduce the number of false negatives of the SafeMath usage determination by adding more compiled SafeMath function bytecodes. Compiling all SafeMath versions with all compatible nightly compiler versions might further increase the accuracy of the results. However, as there are 13 times as many nightly versions as stable versions, that approach has a lot of overhead. One could also try to compile the libraries with with versions that are not compatible with the `pragma` header in the source code of the library. Finally, more heuristics could be investigated in order to get a better estimation of the compiler and library versions. Possibly, there are other distinct features in the code that are unique for other compiler versions, not necessarily only in the header of bytecodes, but for example in the function dispatcher code.

One application of our compiler version estimation could be to find vulnerable smart contracts. The developers of the Solidity compiler maintain a list of known bugs for older compiler versions [59]. It might be feasible to test contracts for which we are certain that they use an older compiler version whether they are vulnerable to one of the compiler bugs. As smart contracts cannot be updated once they are on the blockchain, these bugs persist. Our compiler version estimation can help to narrow down the list of possible bugs for each contract.

Lastly, analysis of the Ethereum system can be extended to encompass not only smart contracts, but also function calls. Metrics of interest would be for example which functions are called most often or the average depth of nested message calls. Further, our library detection could be extended to other libraries and the usage of more ERC standards could be investigated.

List of Figures

List of Tables

Bibliography

[1] R. Kurzweil. “The law of accelerating returns.” In: *Alan Turing: Life and legacy of a great thinker*. Springer, 2004, pp. 381–416.

[2] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008.

[3] V. Buterin et al. “A next-generation smart contract and decentralized application platform.” In: *white paper* (2014).

[4] A. R. Hevner, S. T. March, J. Park, and S. Ram. “Design science in information systems research.” In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[5] C. Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Walter de Gruyter, 2018.

[6] G. Wood. “Ethereum: A secure decentralised generalised transaction ledger.” In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[7] U. Chohan. *A History of Bitcoin. SSRN*. 2017.

[8] C. Decker and R. Wattenhofer. “Information propagation in the bitcoin network.” In: *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*. IEEE. 2013, pp. 1–10.

[9] N. v. Saberhagen. *CryptoNote v 2.0*. 2013.

[10] Complexity Labs. *Evolution of Blockchain*. `https://complexitylabs.io/evolution-of-blockchain/`.

[11] UK Government Chief Scientific Adviser. *Distributed Ledger Technology: beyond block chain*. 2016.

[12] S. Popov. *The Tangle*. 2018.

[13] L. Baird. *Hashgraph consensus: fair, fast, byzantine fault tolerance*. Tech. rep. Swirlds Tech Report, 2016.

[14] S. Rouhani and R. Deters. “Performance analysis of ethereum transactions in private blockchain.” In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. Nov. 2017, pp. 70–74. DOI: `10.1109/ICSESS.2017.8342866`.

[15] M. Fröwis and R. Böhme. “In Code We Trust?” In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017, pp. 357–372.

[16] Ethereum Foundation. *JSON RPC*. `https://github.com/ethereum/wiki/wiki/JSON-RPC`.

[17] J. Pearce. *UML class diagrams as a conceptual models.* `http://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/domain/domainModels.htm`.

[18] J. Evermann. *Using UML for conceptual modeling.* 2002.

[19] L. J. Thomas. *Blockchain Illustrations.* `https://github.com/4c656554/BlockchainIllustrations/blob/master/Ethereum/EthBlockchain5.svg`.

[20] J. Pfeffer. *EthOn – introducing semantic Ethereum.* `https://media.consensys.net/ethon-introducing-semantic-ethereum-15f1f0696986`.

[21] J. Payette, S. Schwager, and J. Murphy. *Characterizing the ethereum address space.* 2017.

[22] R. Norvill, I. U. Awan, B. B. F. Pontiveros, A. J. Cullen, et al. "Automated labeling of unknown contracts in Ethereum." In: (2017).

[23] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. "Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact." In: *arXiv preprint arXiv:1703.03779* (2017).

[24] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou. "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology." In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web.* International World Wide Web Conferences Steering Committee. 2018, pp. 1409–1418.

[25] P. Hegedus. "Towards analyzing the complexity landscape of solidity based ethereum smart contracts." In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).* IEEE. 2018, pp. 35–39.

[26] L. Kiffer, D. Levin, and A. Mislove. "Analyzing Ethereum's Contract Topology." In: *Proceedings of the Internet Measurement Conference 2018.* ACM. 2018, pp. 494–499.

[27] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. "Ethereum query language." In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain.* ACM. 2018, pp. 1–8.

[28] B. Mueller. "Smashing ethereum smart contracts for fun and real profit." In: *HITB SECCONF Amsterdam* (2018).

[29] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. "Erays: Reverse Engineering Ethereum's Opaque Smart Contracts." In: *27th {USENIX} Security Symposium ({USENIX} Security 18).* 2018, pp. 1371–1385.

[30] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. "Vandal: A scalable security analysis framework for smart contracts." In: *arXiv preprint arXiv:1809.03981* (2018).

[31] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. "Formal verification of smart contracts: Short paper." In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security.* ACM. 2016, pp. 91–96.

[32] Y. Hirai. “Defining the ethereum virtual machine for interactive theorem provers.” In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 520–535.

[33] W. Chan and A. Olmsted. “Ethereum transaction graph analysis.” In: *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE. 2017, pp. 498–500.

[34] D. Ron and A. Shamir. “Quantitative analysis of the full bitcoin transaction graph.” In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 6–24.

[35] M. Fleder, M. S. Kester, and S. Pillai. “Bitcoin transaction graph analysis.” In: *arXiv preprint arXiv:1502.01657* (2015).

[36] T. Pham and S. Lee. “Anomaly detection in bitcoin network using unsupervised learning methods.” In: *arXiv preprint arXiv:1611.03941* (2016).

[37] A. Bogner. “Seeing is understanding: anomaly detection in blockchains with visualized features.” In: *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers*. ACM. 2017, pp. 5–8.

[38] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. “BlockSci: Design and applications of a blockchain analysis platform.” In: *arXiv preprint arXiv:1709.02489* (2017).

[39] B. Arvanaghi. *Reversing Ethereum Smart Contracts*. `https://arvanaghi.com/blog/reversing-ethereum-smart-contracts/`.

[40] F. Sakharov. *Reversing EVM bytecode with radare2*. `https://blog.positive.com/reversing-evm-bytecode-with-radare2-ab77247e5e53`.

[41] Ethdocs. *Account Types, Gas, and Transactions*. `http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html`.

[42] Ethereum Foundation. *Design Rationale*. `https://github.com/ethereum/wiki/wiki/Design-Rationale`.

[43] Ethereum Foundation. *The Solidity Contract-Oriented Programming Language*. `https://github.com/ethereum/solidity/`.

[44] Read the Docs. *Libraries*. `https://solidity.readthedocs.io/en/v0.4.25/contracts.html#libraries`.

[45] Read the Docs. *LLL Introduction*. `https://lll-docs.readthedocs.io/en/latest/lll_introduction.html`.

[46] OpenZeppelin. *OpenZeppelin is a library for secure smart contract development*. `https://github.com/OpenZeppelin/openzeppelin-solidity`.

[47] E. Medvedev. *Exporting and Analyzing Ethereum Blockchain*. `https://medium.com/@medvedev1088/exporting-and-analyzing-ethereum-blockchain-f5353414a94e`.

[48] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber. "New kids on the block: an analysis of modern blockchains." In: *arXiv preprint arXiv:1606.06530* (2016).

[49] V. Saini. *Getting Deep Into Ethereum: How Data Is Stored In Ethereum?* `https://hackernoon.com/getting-deep-into-ethereum-how-data-is-stored-in-ethereum-e3f669d96033`.

[50] Wanderer. *Exploring Ethereum's state trie with Node.js.* `https://wanderer.github.io/ethereum/nodejs/code/2014/05/21/using-ethereums-tries-with-node/`.

[51] M. E. Newman. "Power laws, Pareto distributions and Zipf's law." In: *Contemporary physics* 46.5 (2005), pp. 323–351.

[52] J. W. Markham. "Front-Running-Insider Trading under the Commodity Exchange Act." In: *Cath. UL Rev.* 38 (1988), pp. 69–71.

[53] C. Dannen. *Introducing Ethereum and Solidity.* Springer, 2017.

[54] Ethereum Foundation. *EIP 20: ERC-20 Token Standard.* `https://eips.ethereum.org/EIPS/eip-20`.

[55] Ethereum Foundation. *EIP 721: ERC-721 Non-Fungible Token Standard.* `https://eips.ethereum.org/EIPS/eip-721`.

[56] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[57] Dexaran. *ERC223 token standard.* `https://github.com/ethereum/EIPs/issues/223`.

[58] Ethereum Foundation. *Current and historical (emscripten) binaries for Solidity.* `https://github.com/ethereum/solc-bin/`.

[59] Read the Docs. *List of Known Bugs.* `https://solidity.readthedocs.io/en/latest/bugs.html`.