Informatics

Clustering of Ethereum Smart Contracts using the Graph Database Neo4j

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschafts Informatik

eingereicht von

Milosh Davidovski
Matrikelnummer 01528513

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Monika di Angelo

Wien, 29. Juni 2020

Milosh Davidovski   Monika di Angelo

Informatics

Clustering of Ethereum Smart Contracts using the Graph Database Neo4j

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Milosh Davidovski
Registration Number 01528513

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Monika di Angelo

Vienna, 29th June, 2020

Milosh Davidovski

Monika di Angelo

Erklärung zur Verfassung der Arbeit

Milosh Davidovski

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Juni 2020

Milosh Davidovski

Kurzfassung

Im letzten Jahrzehnt hat der Begriff Blockchain aufgrund des Medienrummels um Bitcoin, der ersten Kryptowährung, immense Popularität erlangt. Bald darauf wurde die Blockchain-Technologie zu einer Inspiration für zusätzliche Anwendungen neben Kryptowährungen. Eine solche Anwendung sind Smart Contracts oder Programme mit dem Ziel, die Vereinbarungen eines Vertrags automatisch und sicher ohne die Unterstützung einer zentralen Stelle auszuführen. Derzeit ist Ethereum die wichtigste Blockchain-Plattform für Smart Contracts.

Smart Contracts im Ethereum-Netzwerk können Teil einer dezentralen Anwendung sein oder als eigene Einheit existieren. Sie können durch eine externe Transaktion (User) oder eine interne Transaktion (einen Smart Contract) ausgelöst werden. Angesichts der Bedeutung und Sensibilität der Informationen und / oder Daten, mit denen Smart Contracts täglich umgehen, ist es wichtig, ein besseres Verständnis dafür zu erlangen, wie Smart Contracts tatsächlich funktionieren, welche Funktionen sie ausführen und wie sie im Ethereum-Netzwerk miteinander verbunden sind.

In dieser Arbeit wird ein Ansatz für das Clustering von Smart Contracts auf Ethereum hinsichtlich der gemeinsamen Funktionalität vorgeschlagen, das die Graphdatenbank Neo4j und andere Visualisierungsmethoden und / oder –werkzeuge verwendet. Es werden verschiedene Datensätze (Partitionen des kompletten Datensatzes an Smart Contracts auf Ethereum), sowie zwei Clustering- Ansätze verwendet, um einen besseren Einblick in die Funktionsweise von Smart Contracts zu erhalten und deren funktionale Ähnlichkeiten zu verstehen.

Abstract

In the last decade, the term blockchain has gained immense popularity due to the media hype surrounding Bitcoin, the first cryptocurrency. Soon after, blockchain technology has become an inspiration for additional applications next to cryptocurrencies. One such application is smart contracts, or programs with the aim to execute automatically and securely the agreements of a contract without the support of a centralized authority. Currently, the main blockchain platform for smart contracts is Ethereum.

Smart contracts in the Ethereum network may be a part of a decentralized application or may exist as a single entity. They can be triggered by an external transaction (a user) or an internal one (a smart contract). Considering the importance and sensitivity of the information and/or data that smart contracts deal with on a daily basis, it is important to gain a better understanding of how smart contracts actually work, what functions they perform, and how they are connected in the Ethereum network.

In this thesis, an approach is proposed for clustering Ethereum smart contracts with regard to the functionality they share by using the graph database Neo4j and other visualization methods and/or tools. Different sets of data are used (partitions of the total dataset of Ethereum smart contracts), as well as two clustering approaches, with a goal of gaining a better insight into how smart contracts work and of understanding their functional similarities.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Decentralized cryptocurrencies were first introduced by Satoshi Nakamoto (pseudonym) in the paper: "A Peer-to-Peer Electronic Cash System" in 2008 [Nak08], with Bitcoin as the first/original cryptocurrency. Nakamoto suggests a solution to the double-spending problem without an intermediate centralized party, with the use of a public and distributed ledger to store transactions. Although it is not clearly mentioned, this distributed ledger will later be known as a blockchain, and this technology will find further usage outside of cryptocurrencies.

Soon after, the idea of cryptocurrencies and blockchain technology went viral. The beginning of the second generation blockchain technology was marked by the introduction of Ethereum [But15a]. While Bitcoin can only support built-in cryptocurrency and value transfer, Ethereum, is the first blockchain platform with a built-in Turing-complete language, which allows users to write smart contracts and decentralized applications.

Smart contracts as a term were first used by Nick Szabo in 1996, in his paper [Sza96]. He describes smart contracts as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises". However, Szabos smart contracts and the smart contracts introduced with the current development in the blockcain technology, are not the same. The latter are not necessarily related to the classical concept of a contract.

Cryptocurrency-based smart contract is a concept built on top of the blockchain technology. They are computer programs that run on the network with the purpose of automating the exchange of digital assets without the need for external trusted authority. That is why another name of smart contracts is "self-executing contracts". The primary purpose of smart contracts is to automate processes. Vitalik Buterin, the co-founder of Ethereum describes smart contracts as "systems which automatically move digital assets according to arbitrary pre-specified rules" [But15a]. However, the larger purpose of a blockchain is to ensure immutability and decentralization. Immutability means that after a smart contract is created, it cannot be changed or modified in any way. And decentralization means that the nodes in the network are not dependent on a single master node, but control is distributed among all nodes.

Currently, three main platforms exist for Smart Contracts, that have been applied on top of the Blockchain, which include Bitcoin, NXT, and Ethereum [AH19]. Ethereum is the major player among the smart contract platforms. So far, it has recorded more than a billion of transactions, among them over 15 million of contract creations, and for that reason, the Ethereum blockchain will be used for the purposes of this masters thesis.

Smart contracts deployed on the Ethereum blockchain may be part of a Decentralized application (dApp). A dApp typically consists of a front-end that interacts with the environment and a back-end that handles the business logic. Some dApps outsource parts of the business logic to a smart contract.

1.2 Problem Statement

Although progress has been made, suitable information on how smart contracts are actually used, which goals they achieve, and how contracts are connected and/or interact with each other, in and out of the scope of a dApp, is scarce. Even when the blockchain data is public, information on contract usage is not readily accessible, but rather has to be extracted, distilled, and analyzed.

When the subject of smart contracts is addressed, Ethereum differentiates itself as the leading platform for smart contracts, with billions of transactions recorded so far, among them millions of contract creations [dAS19]. The blockchain itself, and the data it contains is a far-reaching source of information, but the fact remains that it is not easily accessible, and furthermore, it is growing continuously. It is safe to say that Ethereum and smart contracts are still a fresh and actively researched topic [But15a]. Over the past few years, numerous scientific publications have arisen, researching the topic. However, given the complexity of the problem, it is not surprising that many questions, especially regarding smart contracts, have remained open. This includes the usage of smart contracts in the Ethereum environment with respect to the decentralized applications they help to implement.

Many decentralized applications in the Ethereum ecosystem deal with sensitive information on a daily bases. Areas that include such data are banking, insurance, health care, real estate, legal services, notary services, tax records, digital identity management, voting, authorship and intellectual property rights, and more. Considering the type of data and information being passed around publicly on a decentralized network, and the amount of trust a user needs to have in the security of the smart contracts, more knowledge about the type of contracts and implementation is necessary.

In order to better understand the relationship between smart contracts and dApps, a closer look at the functions that deployed contracts implement is needed. When representing contracts as bipartite graphs where the contracts themselves are one group of nodes with only connections to the second group of nodes, being the functions they implement, graph algorithms can be leveraged in order to find structures in this particular graph. The thesis aims at answering the following research questions:

- RQ1: Without prior knowledge about the contracts, which structures/ patterns can be observed in the contract graph?
- RQ2: With prior knowledge about the contracts, which structures/ patterns can be observed in the contract graph?

1.3 Aim of the Work

The overall aim of the work is to gain a better understanding of smart contracts in the Ethereum ecosystem, regardless of whether they are a part of a decentralized application or live as a single entity in the network. Furthermore, this thesis will be of service to the continuous and ongoing work and research on the analysis of smart contracts and the Ethereum blockchain.

In more detail, the expected results comprise the following two parts, corresponding to the two research questions defined in the previous section:

- Without using prior knowledge, we expect to produce clusters of Ethereum smart contracts with respect to the functionality they share. Depending on the clustering approach chosen, clusters of different sizes and density are expected.
- When using prior knowledge, in the form of labeling known smart contracts, an improvement regarding the quality of the clustering is expected. Smart contracts that are known to be similar, or share some functionality are expected to gravitate towards each other.

The two research questions, and their expected results, are similar. Meaning, that if quality clustering is produced in the first part, when no knowledge is applied, clearer and more concise results are expected for the second one.

1.4 Methodology

Given the complexity and sensitivity of the topic, two separate research methods will be required to properly answer the research questions specified in section section 1.2, a systematic literature review and a graph analysis of the smart contracts in the public Ethereum network. In order to provide a better overview, the methodological approach is split and listed below in several steps.

1.4.1 Systematic Literature Review

The first step for a paper of this kind is to create a systematic literature review. Okoli in [OS10] proposes a systematic literature method in information systems. This method is comprised of four phases, which are explained in the following subchapters.

Planing

This phase includes specifying a purpose and designing a protocol of the review. The first question that needs to be answered is why a literature review is needed, and what is the purpose of conducting one [OS10]. The purpose also includes a draft of the research questions the literature review will help answer. By protocol for the literature the author of [OS10] means designing a roadmap towards the answers to the research questions of a thesis. Creating a protocol is an imperative part of conducting a systematic literature review and aims to minimize bias in the study by defining how the review will be conducted [BKB+07].

In regards to this thesis, in this phase, the purpose of the systematic literature review is two-fold. First, to gain more insight into the topic, as a relatively fresh and still evolving area, as well as to provide the theoretical basis required for the completion of this thesis. General literature research regarding smart contracts and blockchains is needed to acquire more knowledge about the rather new fields of smart contracts and blockchains, as well as literature research for topics outside of the area of cryptocurrency, blockchain technology, and smart contracts, but still important for the completion of this paper. Such topics include graph databases, clustering, and similarity algorithms. And second, a more detailed approach is needed, in regards to smart contract analysis, analysis of decentralized applications, and blockchain graph analysis. It is important to note that the literature research will be primarily focussed on Ethereum, as the main smart contract platform for this thesis. However, papers and readings on Bitcoin and other blockchain platforms are also required, to further understand the blockchain technology and smart contracts, and a comparison between the platforms. Hyperledger Fabric, Nem, Stellar, and Waves are regarded as some of the best smart contract platforms [Dav20], besides Ethereum, and present good candidates for this part of the research.

Selection

The selection phase includes the details of the literature review in regards to "applying of practical screening" and "searching for literature" [OS10]. The former requires the authors to be specific in which studies are to be included in the research and which are to be disregarded without further examination. The later, however, need the authors to be explicit in the details of the literature search.

In the second phase, in regards to this thesis, the selection process will be implemented in the following manner:

- The database for the selection of academic articles and papers will be a combination of ACM Digital Library, IEEE Xplore Digital Library, and Google Scholar.

- As a time frame for the selection of papers the years between 2013 and 2019 are chosen because smart contracts as a technology on top of blockchain are a new technology in the field of information systems.

- In the search for articles and papers, the following key words and phrases were used: "Smart contracts", "Ethereum smart contracts", "Blockchain", "Smart contracts in decentralized applications", "Smart contracts analysis", "Smart contracts graph analysis", "Blockchain graph analysis", "Ethereum graph analysis", "Bitcoin graph analysis", "Ethereum decentralized applications", "Decentralized applications", "Blockchain decentralized applications", and further variations of these queries.

Extraction

This phase is comprised of data extraction, exclusion and inclusion criteria, and quality of the selected literature. The specific material that is relevant to the thesis needs to extracted from the selected literature in the previous phase and used as raw material. Furthermore if in the previous phase the partial screening was subjective and based on the relevance to the topic being researched, in this phase the quality of the papers plays an important role.

In this phase papers and articles that are published outside the time frame of 2013 - 2019, as well as papers that are not accessible or could not be downloaded, and books that need to be bought, were excluded from the research. So were papers in which it is clear to see that the content does not suit the purpose of the thesis. A prominent example is the literature found for the search query "decentralized applications", which produces many results centered on the business ideas without putting any focus on the technical details.

Execution

In this phase, the papers have been screened, selected, and scored based on its quality, they need to be combined in order to make a comprehensive sense out of their number, which is often large [OS10]. The final step, in this phase and in the scientific literature review method as well, is to write a review.

Regarding this final step in the thesis, as explained above, all relevant papers and articles need to be combined, compared, and synthesized, for a better overview of the subject. Due to the novelty of blockchain technology and especially smart contracts, the expected number of papers that need to be synthesized is not enormous, which will make this step easier. However, that also means that another research method is required to gain more knowledge.

1.4.2 Graph Analysis

In this part of the thesis, a systematic study on Ethereum will be conducted using a graph analysis of smart contract in the public Ethereum network, similar to the methods chosen by the authors of [CO17], [CZL$^+$18], and [MF14]. In these papers the authors use the following steps:

- Extract data from the public blockchain network of their choice. The data extracted can be different, depending on the needs of the paper. In the chosen references the authors use Ethereum/Bitcoin transaction, smart contract creatin, and smart contract invocation.
- Persist the data extracted in a database, in order to easily use that data in a graph view for better visualization.
- Additional knowledge can be applied, for a clearer and more insightful visualization of the results. For example, Chan and Olmsted in [CO17], who in their paper try to deanonymize the Ethereum blockchain, use addresses that are known to be associated with hacks from Etherscan.
- Evaluation of results.

Similar as in the method outlined above, in regards to this thesis, the graph analysis will be implemented in four separate steps:

- **Aquisition of data.**

  First of all, Google provides a public data set for Ethereum smart contracts on [clo].

  Next, if the deployed smart contract bytecode was produced by the Solidity compiler which implements the Application Binary Interface (ABI) specification [abi19], function entry points can be reverse-engineered for known function signatures. Thus, for each contract a list of contained ABI function entry points is reconstructed, (4 bytes) that can be called by other addresses. The lists of 4 bytes are used to characterize the deployed contracts. This ABI specification is further explained in section 2.5.

  Finally, the list of function signatures, along with the smart contract's bytecode represents the first data set used for the analysis, or the starting point for the practical part of this thesis.

- **Persisting smart contracts in the graph database Neo4j.**

  Extracted data from the Ethereum blockchain will be persisted in Neo4J database [neo16a]. In blockchain analysis, a lot of knowledge and information comes not just from the data itself, but also from the relationships and connections between the data, and graph databases are designed to treat the relationships between data as equally important as the data itself. Neo4J, being the most popular, scalable, and easy to use graph database, was the choice for this research.

Bytecode and function signatures (as explained in the previous step) will be represented as nodes, and the edges between them will exist if a function signature is contained in the smart contract bytecode, or $(bytecode) \rightarrow (signature)$. Two different sets of data will be used. First, no additional data will be used for clustering. And in a second step, contracts with verified source code and known information on their functionality will be labeled accordingly to improve clustering results.

- **Clustering using graph algorithms.**

  After the data set is persisted in Neo4j, similarity algorithms will be applied, in order to create clusters of smart contracts in regards to the functions they share. The two similarity algorithms that will be used are Jaccard similarity and Overlap similarity, both of which are explained by Bollegala in [sim07], with the purpose of better measuring the semantic similarity between clusters. Both similarity algorithms, the reason why they were chosen, and how they are implemented in regards to this research is further explained in section 5.2

- **Analysis of results.**

  The final step is to analyze the different sets of clusters from the previous steps. After measuring the semantic similarity of the smart contracts clusters with the algorithms described above, it is necessary to make a comparison between the results which will further prove the precision of the research.

1.4.3 Evaluation of results

The final step is to analyze the results of the different methods, and provide answers for the research questions defined in section 1.2.

In regards to this thesis, the results from the systematic literature review will be of use to gain a deeper understanding of Ethereum smart contracts, find and analyze related research, and provide insight on known or labeled smart contracts, which will be especially of use to reseach question 2. The graph analysis will be used to visualize and evaluate the clusters of Ethereum smart contracts produced by the two different approaches.

1.5 Structure of the Work

The rest of this thesis is organized in the following manner. Chapter 2 provides a general introduction on blockchains and smart contracts. As a reference to how the blockchain technology works, the Bitcoin and Ethereum platforms are used as examples, as the two most prominent blockchain platforms. Particular emphasis is put on how the blockchain works and its properties, Ethereum and smart contracts, and smart contract usage and applications.

Chapter 3 gives an overview of the related work in two parts. The first part covers graph analysis of blockchain platforms, for both Bitcoin and Ethereum. The second part focuses on the topic of smart contract classification and understanding and/or labeling unknown ones.

Chapters 4 and 5 cover the design and implementation of our model for clustering Ethereum smart contracts in the graph database Neo4j. They include the used technologies, datasets, possible challenges and solution options, and an overview and implementation of the proposed project architecture.

In chapter 6, an evaluation of the results is provided. This chapter includes a combination of quantitative and qualitative analysis in order to properly give answers to the predefined research questions.

Finally, chapter 7 concludes the work with a summary, limitations of the research, a discussion in regards to the research questions, and provides an outlook on future work.

CHAPTER 2

Background

In this chapter, we introduce blockchain basics, smart contracts and especially Ethereum smart contracts, and decentralized applications.

2.1 Blockchain Fundamentals

A Blockchain is a distributed ledger on a decentralized peer-to-peer network, used to store published transactions in chronological order. First introduced with the arrival of Bitcoin, the main idea of the technology was to present a solution for the double-spending problem, which is a phenomenon where the same digital currency can be spent twice (or more) before the system registers the transaction. Bitcoin, along with its blockchain intended to be "an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party [Nak08]". Blockchains differ between platforms, and for the purpose of this chapter, the blockchain of Bitcoin and Ethereum will be examined. Bitcoin's blockchain as the first example of one of the most simple ones, and Ethereums as the most used blockchain platform for smart contracts and the topic of this thesis.

From a technical perspective, the blockchain can be seen as a state transition system, where every state consist of the ownership status of the cryptocurrency and every transaction causes a transition of the state [But15a]. In this context, transactions are nothing more than a proposition to change the current state of the system. An example of a transaction would be a transfer of cryptocurrencies supported by the platform or tokens. Contract invocations are also examples of a transaction. A transaction must be authorized by the account holder, via digital signature, and verified by all nodes in the network. A digital signature implies that the transaction is authenticated with the private key of the owner of the account [Her19].

Transactions are grouped in blocks for efficiency, in a data structure called Merkle tree [Mer80] (in Bitcoin and Ethereum at least), which contains the hashes of all valid transactions in the block. In figure 2.1, the authors of [JCw+18] show a simple visual representation of blockchain. As seen in the figure, a traditional blockchain contains a header and a body. The header is comprised of metadata, like a timestamp, the Merkle tree root, the hash from the previous block, and the nonce. The body contains the actual data, which are the transactions. Furthermore, the figure shows visually how the Merkle root is derived, representing the hash of all the hashes of all the transactions.

One of the key features of blockchain is cryptographic hash functions, which have the purpose of mapping the data of a block to a smaller size and establishing links between blocks. Each block is identified by a single cryptographic hash. When a new successful block is added to the chain, the calculation for a hash of the current block also takes into consideration the hash of the previous one.

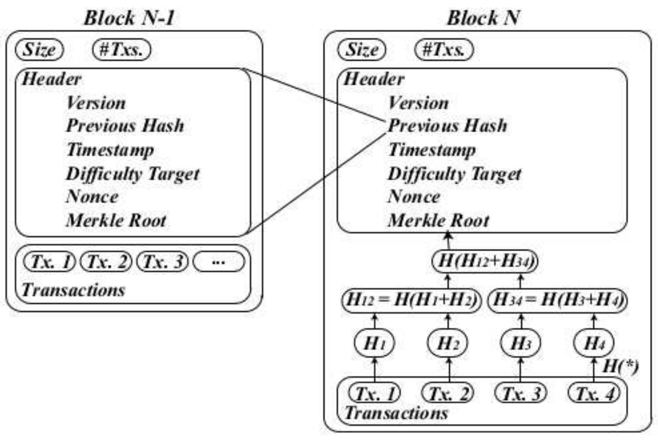


Figure 2.1: A simple version of a blockchain [JCw+18]

2.2 Consensus algorithms

When dealing with a peer-to-peer decentralized network, where each node has it's own copy of the current state of the blockchain, and new blocks need to be appended constantly, there is one question that must be asked. Which node gets to write the new block in the chain, and how is that choice made?

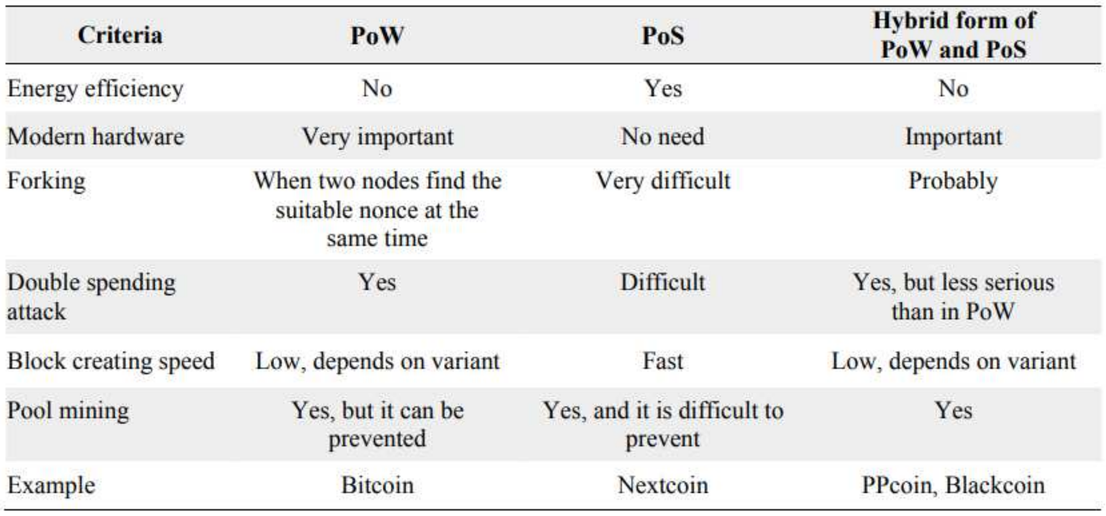
The nodes in a blockchain network, called miners, collectively and repeatedly run a protocol, called consensus, to select which block to append to the ledger [Her19]. There are several algorithms to achieve consensus in a blockchain, and different platforms might use different approaches.

One of the most common and widely used consensus approaches, implemented by both Bitcoin and Ethereum is **Proof of Work (PoW)**. Miners, or users participating in the blockchain mining process have to solve complex puzzles to earn the right to write the new block in the blockchain. This puzzle requires finding a secret value, below a certain threshold, which is a hash starting with a predetermined number of zeros. In this process, two variables are of importance, the nonce and the difficulty. The nonce is the variable that changes until the correct value is found. All nodes change the value of the nonce constantly until it is communicated in the network that the puzzle has been solved, and validated by all nodes. That being said, there are some limitations to the PoW approach. First, PoW is considered to be unfair. As mentioned before, the miners need to solve a cryptographic puzzle, but the speed of finding a solution depends highly on the machine of the user. Modern and expensive machine have a big advantage in this process, and ones in poorer conditions have no or minimal chances in this "competition". And second, the mining process requires a lot of computational power and is very expensive. Even for users with suitable machines it's sometimes not worthwhile to participate. Considering the limitations of Pow, in order to persuade users to participate in the mining process, blockchain platforms introduce incentive mechanisms, where the miner that solves the puzzle is rewarded with the crypto value of the platform. The benefits of this reward mechanism are two-fold. First, it motivates users to participate in the mining process, and second, the cryptocurrency gains value regarding its exchange rate to fiat money.

Considering the limitations of PoW, some blockchain platforms try a different approach, or a **Proof of Stake (PoS)** algorithm. As the name suggests, the key role in the decision of which user gets to write the new block is the current stake of the user in the platform. The motive behind this approach is the assumption that the bigger a user's stake in the system, the less likely it is that the user is malicious. If a user with a significant stake in the network is malicious, and their actions impact the system negatively by reducing the value of the cryptocurrency, has the most to lose. An example of a public platform using PoS is Nxtcoin [whi16]. If a miner owns a coins and the total amount of the remaining coins is b, then the odds of mining the next block are exactly a/b. The disadvantage of unfairness is an issue in this approach as well, due to the fact that miners that own a large piece of the stake have better chances, or a "rich getting richer problem".

Several authors in their respective papers ([KN12], [DFZ16], [BLMR14]) propose examples of **hybrid** algorithms, which use both PoW and PoS. For example in [KN12], the authors and creators of PPcoin introduce the concept of "coin age", which is calculated by stake multiplied by the time that the miner has owned it. Duong in [DFZ16] proposes a variation of this hybrid approach as a solution to the 51 % attack. Betonov in [BLMR14] proposes an approach called Proof of Activity, which also combines PoW and PoS. Other authors propose different variations of this hybrid approach as well, but the implementation behind their versions is always similar.

In figure 2.2, a comparison between these three approaches can be observed. There are also other consensus algorithms that are used. An example is **Proof of Authority (PoA)**, in which new blocks are validated by "approved" accounts. PoA is a common approach in private blockchains [Sun18].



Criteria	PoW	PoS	Hybrid form of PoW and PoS
Energy efficiency	No	Yes	No
Modern hardware	Very important	No need	Important
Forking	When two nodes find the suitable nonce at the same time	Very difficult	Probably
Double spending attack	Yes	Difficult	Yes, but less serious than in PoW
Block creating speed	Low, depends on variant	Fast	Low, depends on variant
Pool mining	Yes, but it can be prevented	Yes, and it is difficult to prevent	Yes
Example	Bitcoin	Nextcoin	PPcoin, Blackcoin

Figure 2.2: A comparison of consensus algorithms [NK18]

2.3 Types of blockchains

In the earliest stages, the concept behind the blockchain was to serve public transactions and be accessible to anyone. That is why, with blockchain, commonly one first thinks of the public type. Today there are three different types of blockchains, depending on the need for the application, the public blockchains, the private blockchains and the consortium blockchains [But15b].

As the name suggests, **public blockchains** are publicly accessible, which means that they are non-restrictive and allow users to interact with the blockchain and read or submit transactions. These types of blockchain are open and transparent, where everyone can review anything and participate in the network. Cryptoeconomics is the combination of cryptography and economic theory to create operating protocols for trust-less and decentralized platforms, which in this type of blockchain will have the role of a substitute for a centralized or quasi-centralized platform [But15c]. The principle is that the level to which someone can influence the consensus process is proportional to the number of economic resources that they can bring to bear.

There are cases, where there is a need for restriction in the network. Here the **private blockchains** are required. Private blockchains are usually used within a centralized organization to store sensitive information about the organization, where all the transactions are visible only to persons who are predefined and are part of the blockchain network. No one outside of the network can access the blockchain and the users' identity is known to every user in the network, whereas the transactions are visible only to those with adequate permissions. With this said, private blockchains are similar in use as public but have a small and restrictive network.

The third type is the **consortium blockchain** where more than one organization is managing a blockchain network, which is why it is semi-decentralized type, and this is the main difference from a fully private blockchain network. A consortium blockchain allows for a certain number of entities to approve and control transactions. Those entities are called consortiums. Consortium blockchains are typically used by banks, government organizations, etc. The consortium blockchain is partly public and partly private, since it provides security which is inherited from the public blockchains, plus restrictions which are inherited from the private blockchain.

2.4 Properties of blockchains

While conducting a scientific literature research on papers on blockchain technology, various different definitions and explanations about the technology were found ([XWS+17], [NH17], [Pil15], [Loo16], [Lem17], [JCw+18]). Some of them focus mostly on the cryptocurrency and transaction side of blockchains, and others state the possibilities of the technology outside of cryptocurrencies. However, in all of them, several properties, in the form of keywords, keep repeating. Some examples are described below:

- **Immutability**

  Immutability is regarded as one of blockchain technology's most important properties and is omnipresent in every paper mentioned above. Immutability in the network is achieved through the consensus protocol of the platform in question and the cryptographic hashing functionality of the blockchain. Once a block is deployed in the network and verified by all other nodes, it is nearly impossible to be modified.

  Jiang and Lemieux in [JCw+18] and [Lem17] respectively, stress the importance of immutability of the blockchain as a means to prevent untrustworthy or malicious modification on records, with Lemieux focussing on recordkeeping systems in general and Jiang on healthcare systems.

  Other authors, like Petersen and Xu in their papers [NH17] and [XWS+17], both place immutability as one of the main property of blockchain. Furthermore, the authors of [Pil15] go one step further, by stating immutability in the system is probably the most essential feature of all.

- **Decentralization**

  There does not exist a central authority or a centralized infrastructure that establishes trust. The network is run completely by its members [NH17]. A copy of the current state of the blockchain is distributed on all machines. After a new transaction is added to the blockchain, its new state is distributed almost instantly to all nodes in the network. All nodes after a certain period must have the same copy of the blockchain.

- **Transparency**

All transactions on the blockchain in a public blockchain platform, are public and can be seen by all users. There are private blockchains, as seen in chapter 2.3), but event there, the data is transparent to the users of that blockchain. Loop in [Loo16] expresses the importance of transparency on retail supply chains, and the possibilities using blockchain technology.

- **Integrity**

  The data is supported by cryptographic techniques and algorithmic rules to check transactions and ensure integrity in the system [XWS+17]. Users sign the transactions/messages with a public-private key hash signature and only the owner can initiate them, as explained in chapter 2.1). However, because keys are not linked to real-world identities, the user in question can remain anonymous [NH17].

2.5 Ethereum and Smart Contracts

With the introduction of Bitcoin, a basic concept of smart contracts was already introduced. In the Bitcoin eco-system, smart contracts are written in a simple stack-based scripting language. The main goal of Bitcoin is to be a cryptocurrency platform, which means the purpose of Bitcoin is the transfer of digital coins. The data a user needs to provide to ensure a verified transaction needs to be implemented by a more complicated script, including a basic public key ownership mechanism [But15a]. However, Bitcoin's smart contracts have several limitations, which include lack of Turing-completeness, value-blindness, lack of state, blockchain-blindness, which Ethereum looks to solve. That is why when using the term "smart contract" in blockchain technology, people usually refer to Ethereum smart contracts.

Vitalik Buterin in [But15a] defines Ethereum as a blockchain that allows users to write smart contracts and decentralized applications, where arbitrary rules for ownership can be defined, as well as transaction formats and state transition functions, thanks to its built-in Turing-complete programming language. Where Bitcoin's smart contracts serve the single purpose of cryptocurrency transfer, Ethereum extends that into much more, by allowing a user to include business logic on the distributed ledger. Ethereum serves as a platform for various decentralized applications built on top of blockchain technology. These dApps typically connect users to the blockchain via a web-based application using a known language like JavaScript or Python [com17].

In figure 2.3 a sample of the Ethereum blockchain is shown. When looking at the blockchain as a data structure, many similarities between Ethereum's and Bitcoin's chain can be observed. Every block comprises a timestamp of the block creation, a nonce used for mining, the hash of the previous block, and a list of transactions. The main difference in the blockchain of these two platforms comes from the transactions. While Bitcoin supports only one type of transaction, currency transfers from one address to another, Ethereum's transactions are more complex. Sometimes a transaction is not really a transaction by its general definition, and that is why a more common phrase in the Ethereum eco-system is **message**, which describes the interactions between accounts [But15a].

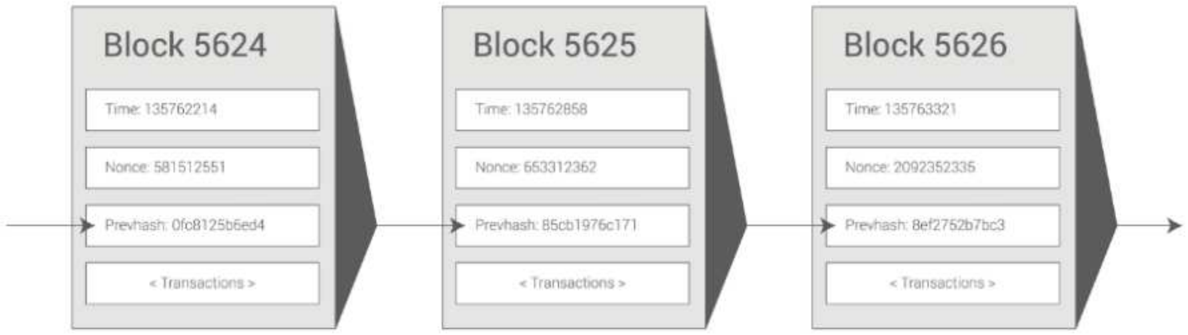


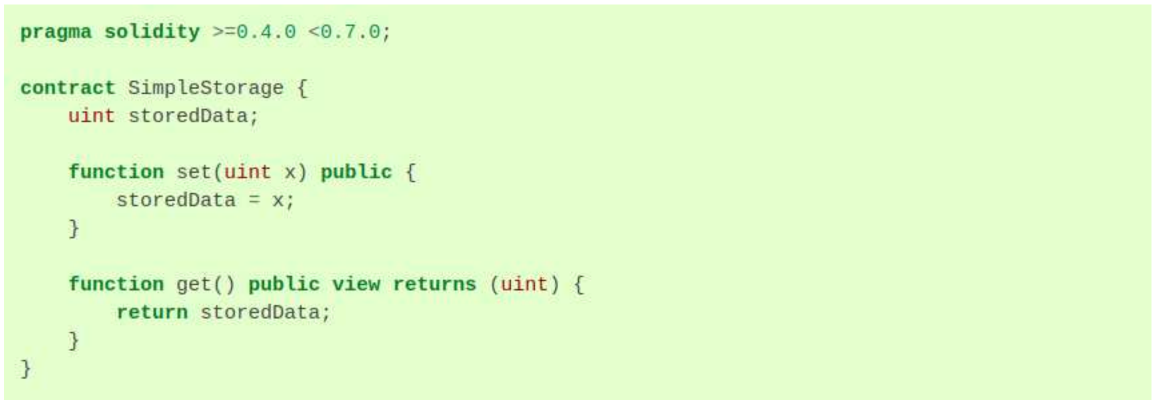
Figure 2.3: Ethreum blockchain example [But15a]

Smart contracts are a key feature in Ethereum. As already stated in section 1.1, smart contracts are computer programs deployed on a blockchain, which when triggered by a transaction or invocated by another smart contract, fulfill a certain purpose, which can vary from a something simple, as a transfer of cryptocurrencies, to a more complicated one. Smart contracts can have many different functions. They can be stand-alone smart contracts, which live as single entities in the blockchain or be part of dApps.

Any users in the network can write smart contracts, by using special, high-level blockchain programming languages like Solidity, Serpent, or Viper, and deploy them to the public network. In figure 2.4 an example code snippet can be seen, with simple get and set methods. This high-level code is later converted to a low-level, stack-based bytecode language, or the Ethereum Virtual Machine (EVM) code, which consists of a series of bytes, where each byte represents an operation [But15a]. Then finally, the smart contract is deployed to the network. Once deployed, the smart contract code can no longer be altered.

This Ethereum Virtual Machine is in the heart of Ethereum and is capable of executing code of arbitrary algorithmic complexity [com17]. Like all other blockchain technologies, Ethereum runs on a decentralized peer-to-peer network, where every node has a copy of the current state of the blockchain. Furthermore, all nodes run the EVM and execute the same instructions, and for this reason, Ethereum is sometimes referred to as a "world computer" [com17].

Unlike Bitcoin's blockchain, which is basically a list of transactions, Ethereum's basic unit is the account. Accounts play a central role in Ethereum and there are two types of accounts [com17]:



```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Figure 2.4: Example Solidity code [com]

- **Externaly Owned Accounts (EOA)**

  EOAs are accounts owned by users. They resemble the only type of account introduced with Bitcoin. EOAs are uniquely identified by addresses of 20 bytes. An account can issue transactions, which in Ethereum are called messages, in order to send Ether and/or data to another EOA or contract account. Also, a message is needed when an invocation or the creation of a smart contract is needed. With each transaction, an account pays a certain transaction fee. In this sense, EOAs can be seen as digital wallets.

- **Contract accounts**

  Contract accounts are basically smart contracts, and they do not belong to a physical person. They are able to send messages between themselves and to EOAs. Unlike EOAs, contract accounts cannot send messages by themselves. They are pieces of code, that when triggered by an external or another contract account, will run and fulfill their predefined purpose no matter what.

Just as Bitcoin, Ethereum has its own cryptocurrency, called Ether. Ether is stored in accounts (both EOA and contract accounts) and can be spent or earned with transactions. However, while Bitcoins are used exclusively for currency transfers, Ether is used also for computational power. Ethereum introduced the concept of **gas**, where a user needs to pay a certain amount of Ether to use the computational power of the network. An example of this would be the deployment of a smart contract. Depending on the complexity, each smart contract, deployed on the blockchain has its own gas price that the creator had to pay. For every operation executed on the EVM, a certain amount of gas is consumed. However, due to the fact that there are numerous smart contracts of different sizes and complexity, and that information is hidden from the users, a single user cannot know the computational power needed for executing a smart contract. In other words, a user does not know how much Ether will a transaction cost. As a solution to this problem, Ethereum introduced a variable called **gas-limit**. When making a transaction, users set this gas-limit, and it represents the maximum amount of cryptocurrency a user is willing to spend for a single transaction. If the execution of a smart contract requires more gas than a user is willing to spend, the transaction is aborted, the EVM stops executing and returns an error.

Apart from the native coin, Ethereum also has **tokens**, which are smart contracts built on top of the native coin of the blockchain platform, and can act as currency themselves. Unlike Ether, which represents a digital currency, tokens may represent a variety of transferable and countable goods. Examples include digital and/or physical assets, shares of a company, memberships, loyalty points, and more [VL19]. Any user on the network can create these smart contracts where they can develop and specify their digital asset. Most tokens on the Ethereum comply with the ERC20 standard [FV15], which is a set of functions and events that must be implemented for a token contract to be considered as compliant with this standard. This standard is not an enforced rule, but is highly recommended to dApp developers so that their tokens can undergo interactions with various wallets, exchanges, and smart contracts without any issues [Ros17].

Smart contracts are invoked, both by and external user and by other smart contracts, by sending a message containing data and/or Ether to the address of the contract. The standard way of interaction with smart contracts is through the Contract ABI [abi19]. For contracts adhering to this specification, the call data contains the called function. This function is specified by its first four bytes, which are the four most significant bytes of the Keccak-256 hash of the signature of the function. "The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e., the function name with the parenthesized list of parameter types. Parameter types are split by a single comma – no spaces are used." [abi19]

In figure 2.5) a list of ERC20-compliant functions can be observed, with the functions names on one side and the 4-byte signatures on the other.

2.6 Smart Contract Applications

What made Ethereum and the introduction of Turing-complete smart contracts different than the rest of the blockchain platforms at that time, was that these technologies went from coin transaction systems to sophisticated systems capable of much more. Over the years, blockchain and smart contracts, because of their unique properties, have found their way in diverse application areas. Examples include notary services, open government, games, finance, law, medical, Internet of Things (IoT), Financial Technology (FinTech), and more. Use cases where smart contracts prove their value are the following:

- **When a trusted third party is required**

  This is the most common and basic use case for smart contracts. Legal contracts may require an external authority to determine if the conditions of that same

Classification			Signature	First 4-byte Keccak hash
ERC20	Required	Method	totalSupply()	`18160ddd`
			balanceOf(address)	`70a08231`
			transfer(address,uint256)	`a9059cbb`
			transferFrom(address,address,uint256)	`23b872dd`
			approve(address,uint256)	`095ea7b3`
			allowance(address,address)	`dd62ed3e`
		Event	Transfer(address,address,uint256)	`ddf252ad`
			Approval(address,address,uint256)	`8c5be1e5`
	Optional	Method	name()	`06fdde03`
			symbol()	`95d89b41`
			decimals()	`313ce567`

Figure 2.5: ERC20-compliant functions [VL19]

contracts have been fulfilled or not. In some cases when a lawyer or a notary is required. Another exemplary use case is a bank for a cash transfer, where the bank acts as an external authority. Or a lawyer and/or notary services in cases of transfer of ownership rights.

- **Initial Coin Offering (ICO)**

  This will be regarded as a separate use case because of how common it is. Initial Coin Offerings is a type of token sale, commonly used by startups to get public funding. By using ICOs startups can cut out third parties, like banks, and raise funding themselves by offering company assets, like shares, in the form of tokens [She19].

- **Keeping records**

  As the blockchain can be seen as a distributed ledger, any case that involves keeping track of records is a possible application. An example would be keeping medical records.

- **Cases when trust in the system is required**

  Due to the properties of blockchain, the risk of fraud is reduced, and users can have more trust in the system, whether it is trust in a vendor (trust in the system) or electronic voting.

CHAPTER 3

State of the Art

This chapter covers related work in the area blockchain and smart contracts analysis and is divided into two separate sections. First, papers on blockchain graph analysis will be examined. These papers focus on topics that use graph databases to persist and analyze different aspects of the blockchain. Some focus primarily on transaction graphs and put the main focus on the communication between accounts, while others focus more on smart contract analysis. The second section comprises related work strictly on smart contracts in the Ethereum network. More precisely, papers on smart contract clustering and classification are examined, as a part of this thesis, especially in regards to research question three.

3.1 Graph analysis of blockchain platforms

When it comes to a graph analysis of the Ethereum platform, two papers differentiate themselves as the most relevant in regards to this thesis. Chan *et al.* [CO17] propose a model to de-anonymize Ethereum, where anonymity is supposed to be guaranteed, by persisting Ethereum transactions in the graph database Neo4j and running different algorithms with Neo4js powerful scripting language Cypher. This paper has significant potential relevance to this thesis and was the reason behind the decision to use Neo4j as the graph database.

In the proposed model, both **normal** and **internal** messages were considered. Normal meaning transactions between external accounts, which are part of the blockchain and are signed by a public key, and internal meaning messages between smart contracts, which are not included on the public blockchain, but a result of a contract being executed. The two types have a certain amount of Ether assigned to them, and that makes both of them important for better results.

Using custom queries, two specific and known addresses were chosen as starting points. They were the addresses of two famous hacks on blockchain systems. The first one was the Gatecoin hack in May 2016 [gat16], and the second was the hack of Daschcoin in July 2017 [coi17]. Starting from here, further addresses were loaded that were associated with transactions. This process was repeated three steps down. Furthermore, some additional logic was implemented. Addresses with a large number of in and out-degree transactions were disregarded because there is a major possibility that those accounts are associated with tumbler service, exchange or gambling smart contracts, which would make the results harder to interpret. Also, ICOs and digital token transfers were ignored and were not considered as part of the implementation. On figure 3.1, the resulting graph can be observed, with the address of the Gatecoin hack selected as a root. The blue circles represent accounts and the green transactions.

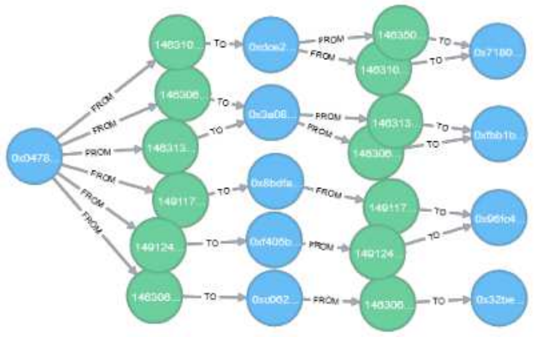


Figure 3.1: Neo4j output for the transactions from the Gatecoin Hack[CO17]

However, this research has some limitations and shortcomings. The main limitation of this graph model was the size and memory needed in order to gain clearer insights. The blockchain data is simply too large to be queried and manipulated in a simple manner.

Another limitation of this approach was that clustering of addresses in Ethereum was not possible. Ethereum's blockchain does not rely on an unspent transaction output to be the input of the next one, which makes the heuristic of clustering accounts owned by the same users impossible.

Chen *et al.* [CZL+18] have conducted the first systematic study of the Ethereum blockchain using graph analysis, by examining both transactions and the smart contract part of the blockchain. The goal of the authors was to gain more insight into the Ethereum blockchain and a better understanding of the Ethereum ecosystem, as well as propose solutions to existing security issues in Ethereum, like attack forensics and anomaly detection.

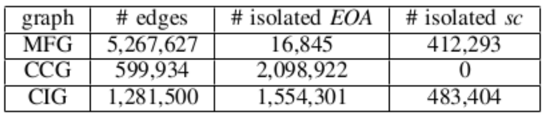
The authors gathered all transactions that happened on the Ethereum network, up to the date of the creation of the paper, and constructed three different graphs to be analyzed, a Money Flow Graph (MFG), a Contract Creation Graph (CCG), and a Contract Invocation Graph (CIG). The methodological approach for this analysis comprises the following steps:

- **Data collection**

  The dataset consists of all published transactions on the Ethereum network, from its launch on 30.07.2015 to 10.06.2017. The total number of gathered transactions in this time period was 19 759 821.

- **Graph construction**

  Three different graphs were created for the purposes of this analysis, namely a money flow graph (MFG), a smart contract creation graph (CCG), and a smart contract invocation graph (CIG). To improve the quality of the data and obtain more fine-grained results, in this second step, the authors also preprocessed the data. Four different types of transactions were excluded from the calculations, namely transactions from one EOA to another when the amount of Ether transferred is zero, a transaction that self destructs a smart contract with no Ether remaining, unsuccessful transactions between EOAs, and unsuccessful smart contract creations. Below, in figure 3.2 the total number of nodes used to create the graphs can be observed, for MFG, CCG, and CIG respectively.



graph	# edges	# isolated *EOA*	# isolated *sc*
MFG	5,267,627	16,845	412,293
CCG	599,934	2,098,922	0
CIG	1,281,500	1,554,301	483,404

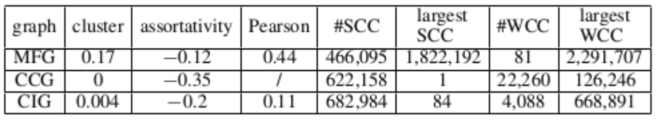
Figure 3.2: Statistics of the three graphs [CZL+18]

- **Graph analysis**

  When it comes to the analysis of the graphs, first three basic values were calculated for every node, the degree, which is the total number of nodes a certain node is connected to, in-degree, and out-degree, which are the number of ingoing and outgoing nodes of a certain node. These metrics are used specifically in directed graphs.

  Next, as can be observed in the table in figure 3.3, seven metrics for the three different graphs are calculated. These metrics include clustering coefficient, assortativity coefficient, Pearson coefficient, number of strongly connected components (SCC), size of largest SCC, number of weakly connected components (WCC), and size of the largest WCC.

  However, it is important to note that isolated nodes were not included in the analysis part of the paper.



graph	cluster	assortativity	Pearson	#SCC	largest SCC	#WCC	largest WCC
MFG	0.17	−0.12	0.44	466,095	1,822,192	81	2,291,707
CCG	0	−0.35	/	622,158	1	22,260	126,246
CIG	0.004	−0.2	0.11	682,984	84	4,088	668,891

Figure 3.3: Metrics of the three graphs [CZL+18]

The is paper covers more topics, including the security of the Ethereum network, but they will not be further mentioned since they are not related to the contents of this thesis.

For this section, papers on graph analysis in Bitcoin were also researched, where the authors propose different models and approaches for graph analysis on Bitcoin's blockchain. For example, Fleder *et al.* [MF14] examine the anonymity of the Bitcoin blockchain, by creating a transaction-graph-annotation system. This system was created in two separate steps. In step number one a system for scraping Bitcoin addresses from public forums was developed. Next, a mechanism for matching users to transactions using incomplete transaction information was included. Bernhard *et al.*[BH16] propose GraphSense, which is a solution that applies a graph-centric perspective on digital currency transaction. It allows users to explore transactions and follow the money flow, facilitates analytics by semantically enriching the transaction graph, supports path and graph pattern search, and guides analysts to anomalous data points. Furthermore, the authors of [BH16] state "The intended solution should be applicable for Bitcoin and any other form of digital currency transactions (e.g. Ethereum)", which makes the paper of significant importance to this thesis.

However, regardless of whether the blockchain of Ethereum or Bitcoin is being researched, one aspect of this type of analysis is almost always the same. The primary focus of most papers addressing this topic is the security and/or anonymity of the blockchain, and that is the main difference from the content of this thesis.

3.2 Classification of smart contracts in the Ethereum network

Papers on smart contract clustering and classification are also relevant to the topic of this thesis. After the contracts have been persisted in the graph database and similarity algorithms have been run, results can be compared with known smart contracts and the clustering can be reaffirmed. Moreover, if the results prove to be reliable, known smart contracts can be labeled and removed from further calculation in order to gain more comprehensive results.

Di Angelo *et al.* [dAS19] examine the temporal and quantitative aspects of smart contracts, with the goal of gaining a deeper understanding of the types of smart contracts on the Ethereum network and their activities. The dataset for this research was comprised of all smart contracts deployed on the Ethereum blockchain up to block 6.9 million, or from its launch to the end of 2018.

By investigating the lifespan and activity patterns, the following types of smart contracts were differentiated in this paper:

- **Loners**

  Loners are smart contracts deployed on the Ethereum network but have never been called. The authors have noticed that these type of contracts comprises 63% of all smart contracts deployed on the network. The total number of loners in the study period is 5 156 658, or 46% of all smart contract creations.

- **Destructed Contract**

  A destructed contract is considered any contract that has executed a self-destruct operation at some point in time. In a single transaction, a smart contract can run a self-destruct operation successfully multiple times. Consequently, the number of self-destruct operations is around 10 times larger than the number of destructed smart contracts. More precisely, there were 2 540 995 destructed contracts in the time of the analysis, and 24 438 879 successfully executed self-destructs.

- **Mayflies**

  Mayflies are smart contracts with an extremely short lifespan. They run a self-destruct operation in the same transaction, in which they were created. The number of recorded mayflies in the time period specified in the paper is 1 856 655. However, it is important to mention that all of these smart contracts were created by only 8 992 distinct addresses, and most of them are smart contracts themselves.

- **Sleepers**

  Sleepers are smart contracts with very long sleep time. By sleep time, the authors refer to the time span between the creation of the smart contract and its first invocation. Furthermore, they distinguish between three different types of sleeper smart contracts. Short-sleepers wait for their first invocation up to 8 days, medium-sleepers from 8 to 25 days, and long-sleepers for more than 25 days.

- **Bonkers**

  Bonkers are defined as smart contracts deployed with useless code. Code is defined as bonkers as the authors state it, whose execution leads to fail, revert, or returns a constant value, without changing the state of the blockchain whatsoever. The number of registered bonker smart contracts at the time of the study was 44 883. However, the resulting number does not include empty non-contracts created by mayflies.

- **Breeders**

Smart contrats in the Ethereum networks, whose only purpose and function are to create other smart contracts, are designated as breeders. As a matter of fact, most smart contracts on the Ethereum blockchain are created by other smart contracts. Below, in figure 3.4 statistics on smart contract creations can be observed. Of all accounts creating contracts, only 17% are smart contracts themselves, and yet they are responsible for 81% of all contract creations.

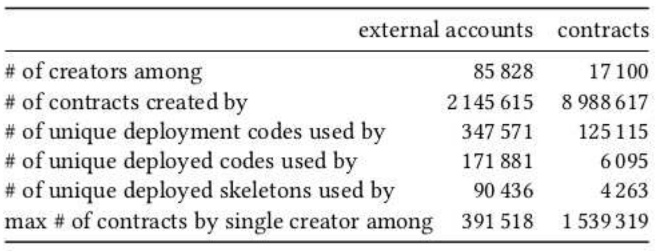
The authors define breeders as smart contracts that are responsible for the creation of at least 1 000 smart contracts. In the period of the study, the number of registered breedes was only 276, and yet they were responsible for 8.76 million smart contract creations.

- **Active contracts**

  The authors define active contracts as smart contracts that have been called at least one time in their lifetime. Two different types of active contracts are recognized in the research.

  The first one is **busy bees**. They are smart contracts that have had at least 1 000 interactions. An interaction is considered any sent or received message, from or to the smart contract. At the time of the research, 27 000 smart contracts of this type were recorded, which is a very small number, considering it “comprises” only 0.6% of all smart contracts. However, these active contracts are responsible for around 505 million smart contract invocations. Moreover, 898 of them are even busier bees with more than 100 000 interactions, and they are responsible for 392 million calls.

  The other type of active contract is **casual worker**, which are smart contracts with less than 1 000 interactions. As can be derived from above, this type of smart contracts comprises more than 99% of all active smart contracts in the Ethereum blockchain.



	external accounts	contracts
# of creators among	85 828	17 100
# of contracts created by	2 145 615	8 988 617
# of unique deployment codes used by	347 571	125 115
# of unique deployed codes used by	171 881	6 095
# of unique deployed skeletons used by	90 436	4 263
max # of contracts by single creator among	391 518	1 539 319

Figure 3.4: Statistics on smart contract creations [dAS19]

Norvill *et al.* [NFS+17] propose a framework to automatically label unknown smart contracts in the Ethereum blockchain by implementing different clustering algorithms on the smart contracts, and conduct both a quantitative and qualitative analysis on the results.

The methodological approach for the completion of this paper was comprised of the following three different steps:

- **Data collection**

  The dataset for this experiment was comprised of all verified smart contracts on the Ethereum blockchain available on `etherscan.io`, up to the time of the experimental process of the research (exact time is not specified).

- **Clustering of smart contracts**

  Two different clustering methods were implemented, namely **Affinity Propagation** [FD07] and **K-medoids** [KMN+02]. The authors applied different distance measurements to determine the similarity of ssdeep hashes of the bytecode of each contract and used the same measurements for both clustering approaches. Three well-known similarity algorithms that were run smart contract bytecode to calculate a distance measurement score that is as precise as possible, namely **Levenshtein**, **Jaccard** and **Sorenson**.

- **Labeling of smart contract clusters**

  In this step, a group of *name words* was generated for each cluster, which was later used to find out the purpose of the smart contracts in the cluster. These name words were acquired with an automated process by extracting the contract name from the page for each smart contract on `etherscan.io`. Next, different naming conventions, such as camel case and snake case, were accounted for. Finally, the four most frequent smart contract names were left as the resulting *name words* to describe each cluster. In figure 3.5 a word cloud can be observed with representing all *name words* it the data set.

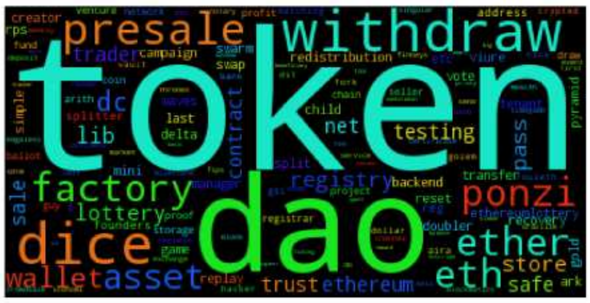


Figure 3.5: Word Cloud for all names in the dataset [NFS+17]

- **Frequency Distribution**

  Finally, a frequency distribution score was given to each of the clusters to measure the coherency and similarity of each cluster. If a is total number of unique name words and b is the total number of name words, the frequency distribution score is calculated as $f(a, b) = 1 - \frac{a}{b}$.

CHAPTER 4

Design

This chapter covers the design of the model for clustering smart contracts in the Ethereum network. For that purpose, the chapter first goes over the idea behind the design, which includes collectiing the dataset of Ethereum smart contracts and Neo4j basics. Next, section 4.1 examines the possible challenges that might occur, along with possible solutions. Finally, in section 4.2, the architecture of the model is displayed and described in detail.

The dataset used in for the analysis of the Ethereum smart contracts was collected from Google Cloud [clo]. Until block 9 500 000, which was at the time of the research, there were 21.8 million deployed smart contracts.

Due to the size of the dataset, certain preprocessing is necessary. First of all, duplicate contracts are removed, because they do not bring any new knowledge to the research. The removal of the duplicates is accomplished in a straight forward manner when we identify smart contracts over the interface they provide. Contracts with identical interfaces are considered to be the same for our purposes - according to the heuristic that they implement the same functionality. The total number of deployed smart contracts then corresponds to 235 770 unique bytecodes, and 74 975 distinct interfaces.

To extract the interface from the bytecode, we relied on the contained ABI function signatures of the contracts, which are already explained in the section 2.5. Each method a smart contract implements has a 4-byte signature, by which it is uniquely identified. Smart contracts can share methods. That means that in the bytecode of two different smart contracts the same function signature can be found, or both contracts implement the same method. As was the scenario with the smart contract bytecodes, these signatures can also be extracted and used in the clustering stage of the thesis. At the time of this research, the number of distinct function signatures in the Ethereum network was 278 103.

It is important to note, that data extraction, compilation to bytecode level, and the complete preprocessing is not part of this thesis, but a ready to be used dataset was already provided.

This relationship $(bytecode) \rightarrow (function\ signature)$, is the main reason why a graph database was chosen. A graph database is a type of database where the relationship between entities is just as important as the entities themselves. By having different types of entities, in the case of this research bytecodes and function signatures, and a relationship between them, again, in this case, wether a bytecode contains a function signature or not, just by persisting them in a graph database, a proper graph structure can be observed.

Neo4j currently differentiates itself as the most frequently used graph database. It comes with its powerful query language called Cypher. Below, an example snippet of Cypher code can be observed which simply returns all nodes, limited to one hundred, so the reader can obtain a more precise picture.

$$MATCH\ (n)\ RETURN\ n\ LIMIT\ 100 \tag{4.1}$$

Cypher is, of course, capable of making more complicated queries for manipulating data, and a more complicated examples will be shown later. Furthermore, Neo4j comes with several "out of the box" algorithms which will prove useful for the next chapters.

4.1 Challenges

From the beginning, two main challenges can be recognized, and both of them are in close relationship to the size of the dataset. One challenge is the amount of memory Neo4j requires to run algorithms and procedures on such a large dataset and the other one is visualization of results. Both are further explored in this section, and possible solutions are proposed.

Memory

Running algorithms in Neo4j is very memory intensive, so working with such a large dataset is impossible, even for a powerful machine. Neo4j has implementations for several similarity algorithms [neo16b], which can be used to calculate clusters. The way these algorithms work, depending on which one is used, is by comparing nodes to each other in regards to certain relationships, or in this specific case, to the function signatures they share.

Following this logic, the number of calculations required to run a similarity algorithm on a dataset would be

$$Number\ Of\ Calculations = \frac{N * (N-1)}{2}, \tag{4.2}$$

where N is the number of nodes on which the algorithm needs to be calculated. Furthermore, considering the number of smart contracts in the dataset after preprocessing is 235 770, this number would add up to 27 793 628 565.

There are several possible solutions to this issue, and some of them are listed below:

- **Vertical or horizontal scaling**

  Vertical or horizontal scaling is probably the simplest method to deal with this challenge. Vertical scaling means running Neo4j on a better machine with more memory and Central Processing Unit (CPU) capacity. And horizontal scaling means distributing the workload to several machines. Both options would solve the lack of memory issue. However, getting more machines or a larger one with more resources is expensive.

  Another problem that might occur with horizontal scaling, is that when graph databases are distributed to two or more machines, several read replicas and only one write replica are created. What that implies, is that when running memory consuming algorithms on the write replica, a bottleneck might still occur, which would make the horizontal scaling futile. Furthermore, one needs to be aware that when distributing graph databases (and NoSQL databases in general) there is always a slight delay while the data synchronizes across all machines.

- **Sharding**

  Sharding, or partitioning, was introduced in the latest version of Neo4j, or version 4.0. This new powerful feature allows user to horizontally scale Neo4j databases limitlessly, in both read and write replicas, with the only constraint being the budget, or the number of machines added [neo20].

  However at the time when this thesis was written, Neo4j version 3.5.14 was used, so this new feature was not experimented with.

- **Integrating Apache Spark with Neo4j**

  Neo4j offers an integration with Apache Spark [neo16c] [apa14], which is a clustered, in-memory data processing solution that scales processing of large datasets easily across many machines.

  When properly integrated into Neo4j, Spark knows how to distribute the workload on multiple machines, which would solve the mentioned problem with horizontal scaling when it comes to read and write replicas. However, access to multiple machines is still necessary. Furthermore, another challenge for this approach is having advanced knowledge of Apache Spark and Business Intelligence (BI) principles.

- **Reducing the size of the dataset and/or algorithm calculations**

  Removing functional duplicates from the unique bytecodes further reduces the data set. This can be achieved by regarding bytecodes as duplicates when they provide the same interface. This yields a reduction from 262 795 unique bytecodes to 74 966 distinct interfaces. A reduction in the number of relationships can be seen in table 4.1.

Another option that would help solve this challenge, is further reducing the dataset or at least partitioning the data in different components and with that reducing the number of similarity calculations. One example is community detection algorithms. Neo4j offers three community detection algorithms, Louvain, Label Propagation, and Weakly Connected Components [neo16d]. These algorithms, as the name suggests, detect communities within the network and partition it accordingly.

Phase	No. Bytecodes	No. Relationships
Deployed Contracts	21 779 683	36 530 349
Unique Bytecodes	235 770	3 682 573
Distinct Interfaces	74 975	1 304 623

Table 4.1: Dataset Information

- **Exporting part of the logic to a Java project**

  The last option proposed is exporting part of the logic to a Java project. If a user is well acquainted with the software, he or she can create a JavaEE or Spring Boot project, and run algorithms on parts of the dataset, while simultaneously combining and exporting results. A prerequisite for this approach is, as was the case with Apache Spark and BI, advanced knowledge of Java EE and/or Spring framework.

Visualisation

Even though Neo4j is powerful when querying the database and can find results relatively fast, its visualization tool is not optimal. The Neo4j Browser and Desktop Graphical User Interface (GUI) allow at most 300 nodes and their relationships to be seen on the screen per result, which means that there is a need for a third party software/tool specially designed for graph visualization.

In order to make the an educated choice, further literature review was conducted on graph visualization, focusing on the following metrics for visualization tool comparison:

- **Performance**

  Due to the size of the data and the purpose of the analysis, a tool is required that is scalable and can handle a huge amount of data for as little memory as possible. Also, performance includes the speed of the network visualization tool for both loading data of different sizes and running algorithms and procedures on the same.

- **Usability**

  Another important aspect is for a tool to be user friendly. The data set will be modified constantly and different cluster similarity and community detection algorithms will be implemented. Because of that, the tool in question will be used constantly with different data. Moreover, dynamically labeling and modifying the data will be necessary.

- **Visualisation**

  Once a graph is loaded and fully visualized, it can be hard for humans to understand and interpret what they see on the screen. For that reason, there exist different algorithms and metrics to make the representation as understandable as possible.

It is important to note that as contenders for graph visualization software were considered only open-source, desktop tools. Several scientific papers have already comparative analyses on different open-source, desktop tools [MSuR15], [PPEKI], [FA18], and the four large network visualisation tools that keep repeating are the following:

- **Cytoscape** [ML12]

  Cytoscape is an open-source network visualization tool primarily used for biological networks and health sciences. Lately, though, it has also found usage as a generic platform and is used for analysis and visualization of more complex networks. Out of the four network visualization tools, Cytoscape has the largest library of additional plugins (>250), and the richest and most efficient collection of algorithms [PPEKI]. However, it does not rank great for large-scale network analysis, and that is its biggest weakness, regarding this research. Also, as Cytoscape is a Java application, it is subject to the memory limitations of Java.

- **Pajek** [HD03]

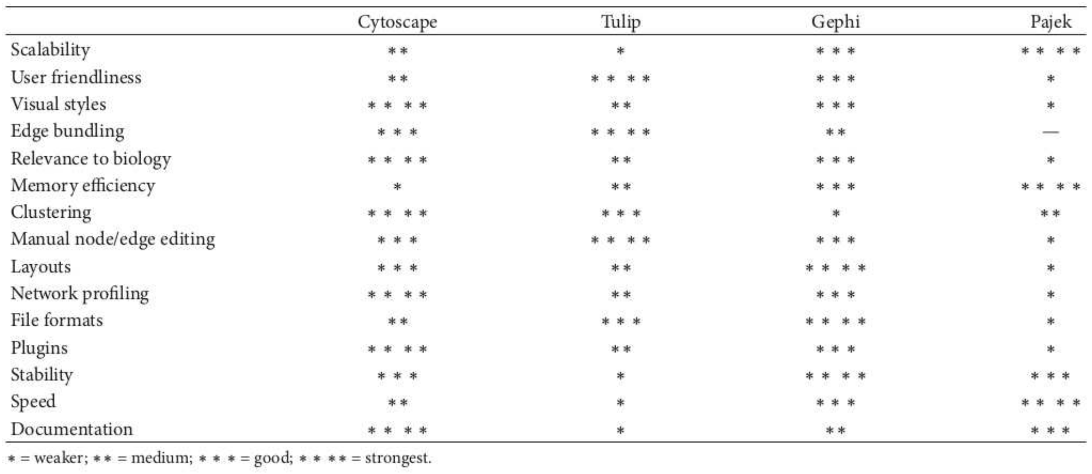
  Pajek is a Microsoft Windows based network visualization and analysis tool and is specifically designed for network analysis of huge datasets. When it comes to scalability and performance, Pajek easily outperforms all other tools on the list and can visualize a million nodes with billions of connections on an average computer. However, Pajek has two main disadvantages. First, even though it is quite strong on analytics, Pajek is relatively weak when it comes to visualization. And second, Pajek is the least user-friendly platform on the list. When it comes to the file formats it accepts, Pajek is very strict, and the best way to communicate to it is trough .net files.

- **Tulip** [LA12]

  Tulip is the easiest-to-use network visualization tool and is recommended for beginners and non-experts. It is packed with a large set of graph-based operations and it is a good choice for medium-sized networks. Tulip is the best option if a dataset needs to be developed or customized dynamically. A user can build his/her own plugins, and modify and visualize the data accordingly [MSuR15]. However, when it comes to large-scale networks it does not perform well. As a matter of fact, out of the four tools in this list Tulip proves to be the worst when it comes to large-scale networks.

- **Gephi** [BHJ09]

  Gephi is the last visualization tool on the list of options, and it is known for both visualization and performing graph-based analysis on a network. Furthermore, it also allows users to interact with the network through on-screen tools in the visualization window. Also important, Gephi has proved to be working flawlessly on averagely configured computers and was able to perform analyses of large datasets, unlike the other tools which can crash sometimes due to low configurations. All three papers have concluded that Gephi has been observed to be the highest overall performer regarding good graphics and maximum functionality.



	Cytoscape	Tulip	Gephi	Pajek
Scalability	**	*	* * *	* * * *
User friendliness	**	* * * *	* * *	*
Visual styles	* * * *	**	* * *	*
Edge bundling	* * *	* * * *	**	—
Relevance to biology	* * * *	**	* * *	*
Memory efficiency	*	**	* * *	* * * *
Clustering	* * * *	* * *	*	**
Manual node/edge editing	* * *	* * * *	* * *	*
Layouts	* * *	**	* * * *	*
Network profiling	* * * *	**	* * *	*
File formats	**	* * *	* * * *	*
Plugins	* * * *	**	* * *	*
Stability	* * *	*	* * * *	* * *
Speed	**	*	* * *	* * * *
Documentation	* * * *	*	**	* * *

* = weaker; ** = medium; * * * = good; * * ** = strongest.

Figure 4.1: An empirical evaluation of the four network visualization tools [PPEKI]

4.2 Architecture

In this section, an architecture of a model for clustering Ethereum smart contracts in Neo4j graph database is proposed, as a solution to the challenges covered in the previous section. In figure 4.2 the proposed design can be observed. It comprises three main components or layers:

- **Data Layer**

This layer consists of two Neo4j databases, one local and one remote. The dataset loaded in graph databases comes from an external data source or a Comma-separated values (CSV) file.

- **Business Logic Layer**

  This layer handles the business logic of the model, more precisely the algorithms run on the smart contract bytecodes and/or interfaces. To that purpose, a Spring Boot project is created and connected with both Neo4j databases.

  It is important to note that part of the business logic is handled by the Neo4j browser/desktop application, or in this particular case, the data layer.

- **Presentation Layer** The presentation layer is completely covered by Gephi [BHJ09], which was already mentioned in the previous chapter, and is further covered in the next subsections, both as a separate tool and in the context of this research.

The following subsections cover these components in more detail, as well as the workflow.

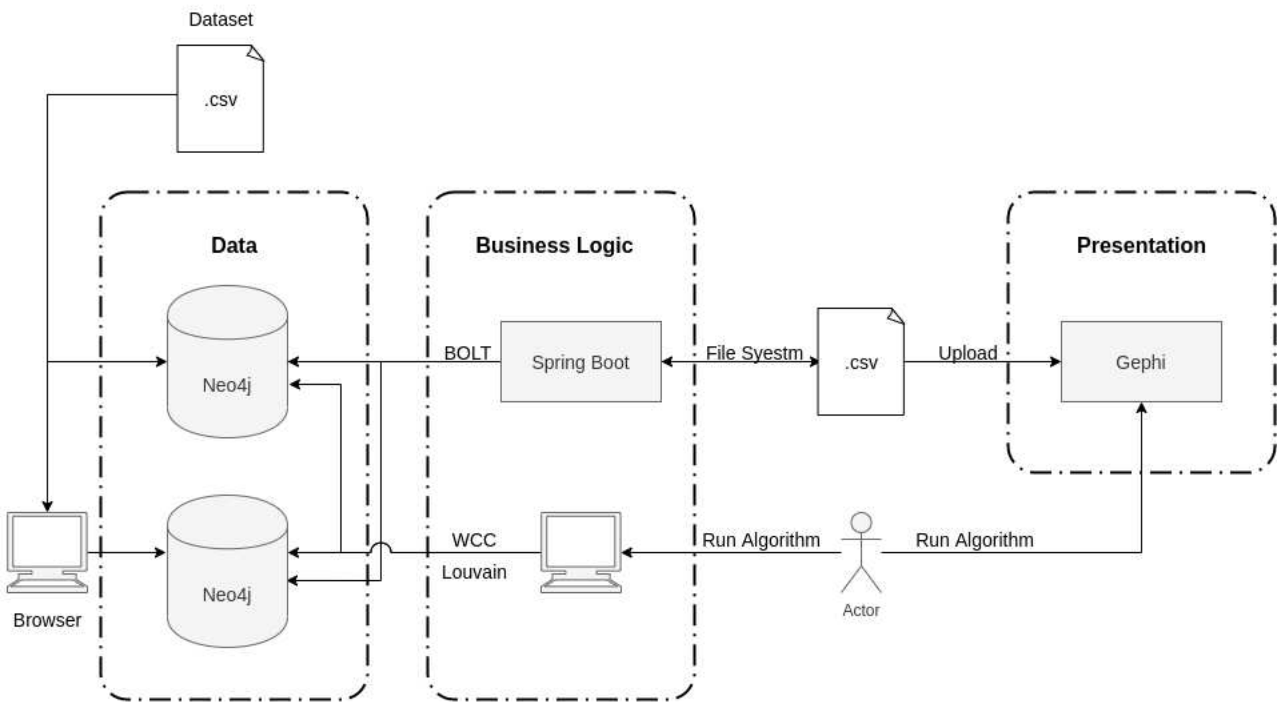


Figure 4.2: Architecture of a model for clustering Ethereum smart contracts using a Neo4j graph database

4.2.1 Data Layer

Neo4j is regarded as the key technology for this thesis. As it was already mentioned in chapter 3, this is not the first time Neo4j was chosen as a database when it comes to Ethereum smart contract analysis, as Chan *et al.* [CO17] make use of its potential, by storing addresses of outgoing and incoming smart contract invocations. However, in the case of this thesis, it is used in a completely different manner. As can be observed in figure 4.2 two different Neo4j databases are used, one local and one remote. The remote database was added at a later point in time because the local machine is not strong enough for more complicated operations. As stated before, even though Neo4j is a powerful graph database, it is very expensive when it comes to memory consumption.

There are two main prerequisites for properly working with Neo4j. The first one is understanding how graph databases work, their purpose, and use cases. To that end, a scientific literature review approach was used. Papers on graph databases and graph database comparison were analysed, such as [JV13] and [DSUBGV+10]. Furthermore, Neo4j offers prominent and always up-to-date documentation [neo16e]. The second prerequisite is a comprehensive theoretical and practical knowledge of the Cypher query language. This second part requires not just basic reading, but technical practice and experimentation with it. One advantage though, for a user that has prior experience with relational databases, is that Cypher has some similarities with Structured Query Language (SQL), which makes the learning process easier.

The simplest way to load data in a Neo4j database is with a CSV file [neo16f]. Cypher offers a *LOAD CSV* command that loads small to medium-sized files to the database. There are several things to know in order to properly load data from a CSV file with this command. First of all, the CSV file has to have a maximum of 10 million records. For a file with more records a different approach must be taken, which is more complex and sometimes using external tools is required. Next, when persisting CSV data in a Neo4j database, the data and import statements need to be handled with care. Labels, property names, relationship-types, and variables are case-sensitive. Finally, when using local files, the CSV files are recommended to be located in a specific import directory. That is because local files are referenced with a $file : ///$ prefix, which specifies the default location of the CSV files.

Once a dataset is properly uploaded, Neo4j, for the most part, works like a normal database, capable of performing all Create, Read, Update, and Delete (CRUD) operations. However, it also implements algorithms and procedures specially designed for graph database usage. For this thesis, two types of algorithms that Neo4j offers are of great importance:

- **Similarity algorithms**

  The goal of similarity algorithms in a graph database is calculating the similarity index between each pair of nodes in regards to their relationships. Neo4j implements six similarity algorithms [neo16b], all measuring the similarity index differently.

These algorithms are used to create clusters of nodes that are similar to each other. In the case of this thesis, similarity measures that are based on the number of shared signatures are of interest. During the scientific literature research, and already mentioned in more detail in chapter 3, Norvill *et al.* [NFS+17] used similarity algorithms with an identical purpose.

- **Community detection algorithms**

  The other type of algorithms that are of interest to this research is community detection algorithms [neo16d]. When referring to a network or graph, a community is considered a subset of nodes within the graph such that connections between the nodes are denser than connections with the rest of the network [RCC+04]. In Neo4j graph databases, nodes in one community will always have a larger number of relationships between each other, than with noded in different communities.

  Neo4j implements three different algorithms of this type, Louvain, Label Propagation, and WCC. Louvain and Label Propagation are non-deterministic by design, and WCC is deterministic. Deterministic meaning that no matter how many times an algorithm is run it will produce the same results, which is not the case with non-deterministic algorithms.

  The purpose of this type of algorithm is to partition the database concerning its relationships. A typical use case for community detection algorithms is when dealing with a huge dataset, which is the case of this research.

4.2.2 Business Logic Layer

The second main component of the proposed solution is the business logic layer of the model and it is in the largest part covered by a Spring Boot project. This approach was chosen for the following reasons:

- The experience of the team with Java EE and Spring Boot applications.
- Spring offers integration with Neo4j databases (**Spring Data Neo4J**), which is very well documented [spra].
- A Java EE or Spring Boot project offers a lot more flexibility when it comes to data manipulation, whether the resulting data is to be exported to another location or kept in memory.

The phrase "in the largest part" is used because Neo4j itself handles some part of the business logic. The Neo4j browser or desktop application is also capable of performing all algorithms and procedures, as well as manipulation of the dataset with custom Cypher queries. A sample query was already presented above on function 4.1. Further, and more complicated queries, as well as usage of Neo4j specific algorithms, are presented in the next chapter. However, all these calculations depend on the memory allocated to Neo4j, and with that, the memory of the machine on which Neo4j runs. That is the main purpose of an extra project or an explicit business logic layer in the architecture.

The main role of the Spring Boot project is to take away some of the memory consumption from Neo4j and add it to Java Virtual Machine (JVM). A reasonable argument would be that with this approach, the only thing that is done, is that the memory consumption is transferred from a Neo4j server to the local JVM of the machine, which again depends solely on the memory capacity of the machine. However, what Java and Spring offer, is more configuration options that reduce memory consumption, as well as different ways of database usage, with the cost being, of course, added complexity in the project.

The benefits of this approach are two-fold. First, with proper configuration and usage, the memory usage of the system can be optimized. And second, by carefully selecting algorithms, one can minimize the number of objects kept in memory at any given time.

This part of the architecture, even though it is considered an extra layer added to reduce memory consumption, is the most difficult one to implement and add to the workflow of the model. On one side Spring needs to connect to the two Neo4j graph databases, prepare the resulting dataset, and export the results in the proper format to the presentation layer on the other. As a consequence of this workflow, several challenges need to be overcome for the proper usage of the component and for getting optimal results.

First, in order to connect to a Neo4j database, Spring uses a Neo4j developed protocol called Bolt [neo15a], which is a lightweight client-server protocol designed for database applications, originally authored by the creators of the Neo4j graph database. Using Bolt and proper credentials, along with further configurations in Spring, a connection can be established with Neo4j. These configurations are further explained in the next chapter.

Next, when working with such a large-scale dataset, the program needs to be able to work continuously, looping over parts of the data and exporting results in a certain format. This process can be done once and will loop through the whole data set. However, this way the machine has to be left to work constantly for hours, maybe even days for some more complicated procedures. A better approach would be to set up a project in a certain way, that the current state is always remembered. Therefore, the program can be stoped at any time for the machine to rest, and continue working at a later point in time without losing the current state.

Finally, the resulting data must be exported in a proper format. Even in this step, certain knowledge is necessary about the following, presentation layer. The formatted and extracted data has to be readable to the graph visualization software of choice, which in the case of this thesis is Gephi. More about Gephi and the presentation layer is further explained in the next subsection.

Below, in table 4.2, the list of technologies used for this project are listed.

Name	Description
Intellij	Integrated Development Environment (IDE)
Kotlin / Java	Programming language
Neo4j	Database
JUnit / Mokito	Testing Framework
Simple Logging Facade for Java (SLF4J)	Logging

Table 4.2: Technologies used

4.2.3 Presentation Layer

The last component in the proposed solution for the thesis is the presentation layer. As in the be observed in figure 4.3, the complete presentation component is covered by a large-scale visualization tool, which in the case of this research is Gephi [BHJ09].

Gephi, as well as three other large-scale network visualization tools, was already briefly covered in section 4.1 as part of the network visualization tools comparisons, when taken into account the memory capacity of the local machine, plus the size and density of the dataset, is the best options.

Apart from the comparison of network visualization tools in the scientific literature research, all four tools were downloaded and sample data was loaded in the to see how they would react specifically with the dataset from this research.

Pajak was disregarded immediately because of its weak results in user-friendliness since it is very specific in the data format it accepts would be a tiresome process. That, along with its poor graph visualization, was the reason Pajak was not included for further experimentation.

In the remaining tools, three sample datasets were loaded to observe their behavior. The sizes of the datasets can be observed below on table 4.3.

Size	Number of Nodes	Number of Edges
Large	16 203	1 100 702
Medium	3 396	12 696
Small	1 024	45 384

Table 4.3: Sample datasets for visualization tool comparison

Once the datasets are loaded, a comparison between all three large-scale vizualisation graph is cunducted, in regards to the tool comparison metrics defined in section 4.1.

Performance

With all three datasets, Gephi performes the best, in both time and scalability. The difference, of course, is most noticeable when the medium or large dataset is used.

Cytoscape performs well when the small and medium dataset is used. However, when the large dataset is loaded on a local, averagely configurated machine, first of all, it takes a long time for the data to load, and second, while running clustering and layout algorithms, the program regularly crashes.

Tulip comes last when it comes to performance, in both time and scalability. The medium dataset takes some time to be loaded in Tulip, and the large one cannot even load without crashing.

Usability

All three graph visualization tools are similar when usability or user-friendliness is in question, with Cytoscape differentiating itself as a little worse than Gephi and Tulip. But only because it uses a lot of external plug-ins that need to be understood, which is can be tedious and time-consuming.

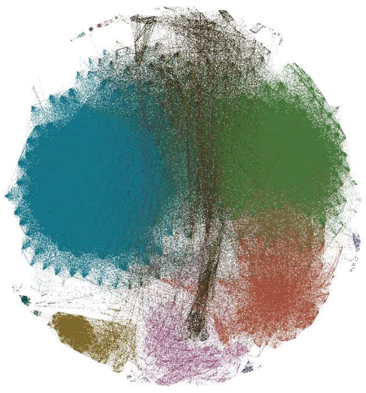
Other than that, all three prove to be easy to use and understandable. Also very important, they are flexible when it comes to the format of the data that can import. The same format of CSV files was imported in all three tools.

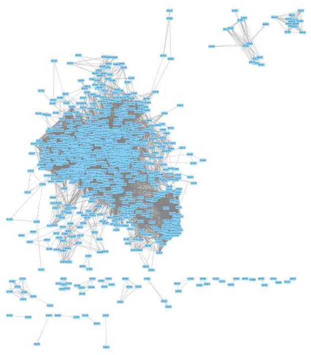
Visualization

In figure 4.3, the resulting graphs can be observed of all three visualization tools with the small dataset loaded. Even from a short glance, on the upper left image, which is Gephi, several clusters can be observed, which are concise and separated by different colors thanks to the modularity algorithm that Gephi implements.

Furthermore, Gpehi scores best in recognizing clusters. As it can be seen in the screenshot, while Tulip and Cytoscape recognize only one large cluster and the reset of the data scattered around, Gephi's force layout algorithm can recognize three. These layout algorithms are further explained in the next chapter.

Finally, it should be noted that because Gephi was chosen as a graph visualization tool for this thesis, further experimentations were made to test its limits. On the machine where the experimentation is being done, with the memory capacity available Gephi can handle up to 20 000 nodes and 1 000 000 edges. When more are loaded the program is unresponsive and might crash. Furthermore, when working with more than 10 000 nodes Gephi slows down considerably and running algorithms on the network are difficult.


(a) Gephi


(b) Cyptscape

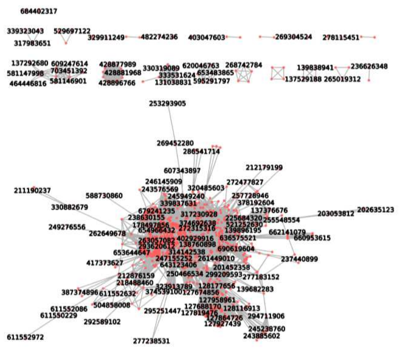

(c) Tulip

Figure 4.3: Result sample comparison for visualization tools

CHAPTER 5

Implementation

This chapter provides a comprehensive description of the implementation of the solution with the chosen architecture and technologies. To that end, code snippets, result outputs, and precise explanations are provided for each of the steps in the implementation process. Similar to how the organization of the architecture section was designed, the following chapters will go into further detail in all three distinct layers of the solution and the workflow that connects them. First, section 5.1 covers the data layer, which includes the set up of the two Neo4j databases, dataset selection and persistence, and initial experimentation with Neo4j and Cypher. Next, in section 5.2, the implementation of the business logic layer is explained in detail. This section covers the integration of a Spring Boot project in the workflow of the model, usage of algorithms and procedures, challenges and how they are overcome, formating and exporting data, and more. Finally, the last section of this chapter, section 5.3, covers our implementation with Gephi, including precise configurations, datasets, and algorithms used.

5.1 Data Layer

As already mentioned in the architecture section and can be observed in figure 4.2, the data layer is comprised of two separate Neo4j databases, one on a local machine and the other one on a remote machine. The remote Neo4j server was provided by TU Wien and has greater resource capacity, to better manage the memory complexity issues of the graph database. Neo4j provides good desktop and browser GUIs, and for this research, the browser version was used and proved to be more than adequate.

Two main steps were taken to ensure proper set-up of the graph databases, with the first being refactoring the data to the correct format, so it is understandable to Neo4j, and the second properly configuring and persisting the data using Cypher. Both will be covered in the following subchapters.

5.1.1 Data Preparation

Neo4j, like any other graph database, recognizes two main entities, nodes and relationships. For the purposes of this thesis, the entities are set up in the following manner:

- **Nodes**

  The set-up of Neo4j, in regards to this research, consists of two different types of nodes. The first type of node is **bytecode**, which represents the smart corntracts, and the second is **signature**, which represents by the ABI function signatures.

  Both nodes are persisted and configured with a single property **id**, on which a unique constraint is added. That means that each node, no matter if it is from type bytecode or signature, will have a property id, by which it can be uniquely identified.

- **Relationships**

  Only one type of relationship is configured in the current set-up of the graph database, and that is the relationship **has**, which implies that for a smart contract b and a function signature s

  $$(b : bytecode) - [: has] \rightarrow (s : signature), \tag{5.1}$$

  if s is contained in b's bytecode.

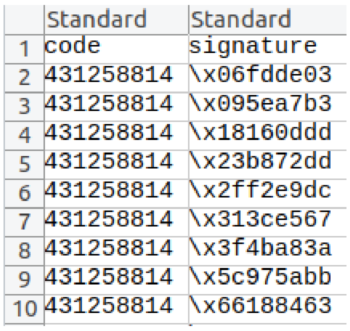
  When this relationship is set up, a directed graph representation is created, of all smart contracts and the function signatures they contain.

As it can be observed in figure 4.2 the method chosen for persisting the data in Neo4j is importing it from a CSV file [neo16f]. Both databases are loaded with the same data from the same CSV file. As is recommended in the Neo4j documentation, the files are kept as simple and concise as possible, with most of the configuration being left to Cypher. The current CSV file imported has two columns, bytecodes and signatures. Below, in figure 5.1 a small sample of the data can be seen.

5.1.2 Loading the dataset in Neo4j

Once the CSV file is properly formatted and located in the correct directory(see section 4.2), the nodes and relationships can be set up using Cypher's *LOAD CSV* function. In the case of this research, the set-up is managed with the Cypher code in the listings:

```
CREATE CONSTRAINT ON (b:bytecode) ASSERT b.id IS UNIQUE
CREATE CONSTRAINT ON (s:signature) ASSERT s.id IS UNIQUE
```



	Standard	Standard
1	code	signature
2	431258814	\x06fdde03
3	431258814	\x095ea7b3
4	431258814	\x18160ddd
5	431258814	\x23b872dd
6	431258814	\x2ff2e9dc
7	431258814	\x313ce567
8	431258814	\x3f4ba83a
9	431258814	\x5c975abb
10	431258814	\x66188463

Figure 5.1: First 10 rows of the data loaded in Neo4j

These Cypher statements create the unique constraints on a property id for both bytecodes and signatures, by which the smart contract bytecode and function signatures can be identified.

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///data.csv" AS line
MERGE (:bytecode {id: toInteger(line.bytecode)})
MERGE (:signature {id: line.signature})
```

These set of statements import the bytecodes and signatures as nodes in the database for a CSV file called "data.csv". The method *MERGE* is used instead of *CREATE* to avoid node duplication.

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///data.csv" AS line
MATCH (b:bytecode {id: toInteger(line.bytecode)}), (s:signature {id: line.signature})
CREATE (b)-[r:has]->(s)
```

These last Cypher statements create the relationships between the nodes. It should be noted that the CSV file needs to be imported separately for both node and relationship creation.

On the second and third function the method *USING PERIODIC COMMIT* 500 is used, which is a method that instructs Neo4j to perform a commit after a certain amount of rows, in this case, 500 rows. This method reduces the memory overhead of the transaction state and is recommended when the CSV file in question contains a significant number of rows.

As a final example in this section, below, in figure 5.2 a sample result from Neo4j can be observed for the following query

```
MATCH (b:bytecode)-[:has]\rightarrow(s:signature)
RETURN b, s
LIMIT 100,
```

which simply returns all nodes for the current dataset loaded in the graph database, limited to one hundred results for greater clarity.

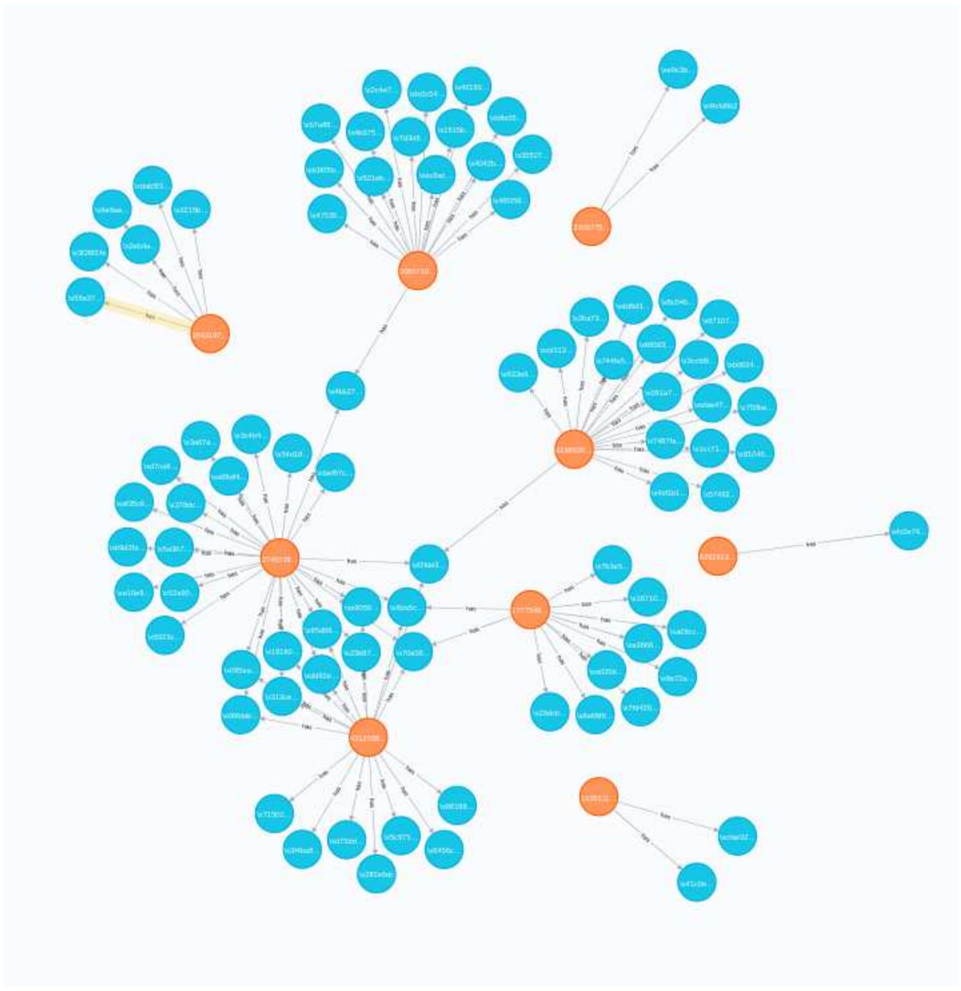


Figure 5.2: Neo4j sample output, returning 94 nodes, more precisely 9 bytecodes (orange) and 85 functon signatures (blue) with 100 relationships "has" between them

5.2 Business Logic Layer

The business logic layer is in the largest part covered by a Spring Boot project. However, that was not the first idea, but came later as a solution to the resource challenges. What started in the beginning as a single local Neo4j database, was later extended to one more remote Neo4j server with more resources, and finally finished with exporting most of the logic to the external Spring Boot project.

The goal of this project is to connect to the two Neo4j graph databases, get the data in partitions, as to not overwhelm the JVM, transform the data and run similarity algorithms, and ultimately export a readable CSV file to the presentation layer. This complete process can be separated into three distinct steps:

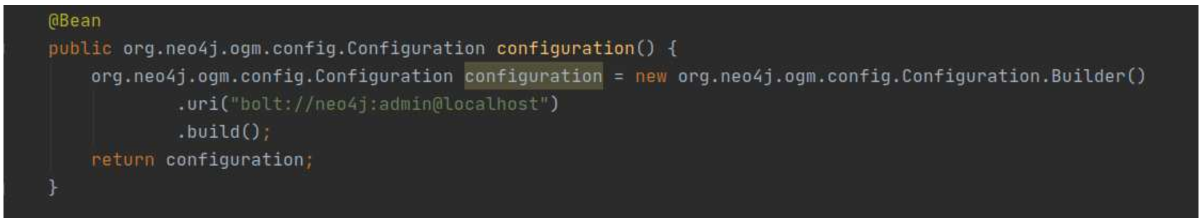
- Setting up the project, which includes installing a maven project, adding required dependencies, and connecting the project to Neo4j.
- Applying proper algorithms and procedures, some directly in the Neo4j Web-application and some in the Spring Boot project, depending on the memory requirements of the algorithms.
- Exporting results to a CSV file, in a format that is optimized and readable for Gephi.

5.2.1 Project Set-Up

Creating a new Maven Spring Boot project is simple. The Spring team offers a project initializer that handles the basics of setting up a Spring Boot project [spr13]. The only additional dependency that is required is the Spring Data Neo4j maven dependency, and it can be added directly through the initializer. Spring Data Neo4j is a project that offers Spring Data support to Neo4j graph databases. It supports Object-Graph-Mapping of annotated Plain old Java object (POJO) entities, Spring Data repositories, transaction handling, and more.

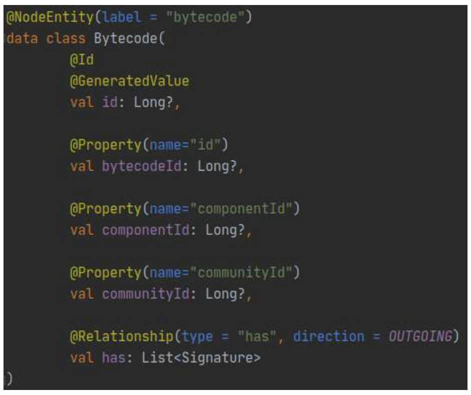
Once the initial set-up of the project is created and the correct dependencies for testing and logging are added, the next steps include connecting to the Neo4j databases and creating the initial version of the repository and mapped entities. In order to connect to a Neo4j database, a driver needs to be configured. Spring Data Neo4j offers three possibilities, Embedded, HTTP, and Bolt. Bolt uses the official Neo4j Java driver and using it requires no extra dependencies to be added, unlike with the other two options. For those reasons and the fact that it is a recommended choice, the Bolt protocol was chosen to connect to Neo4j. Below, in figure 5.3, a code snippet can be observed for a configuration with a Bolt protocol to connect to a local Neo4j database.

Next, the nodes and relationships from Neo4j need to be mapped to entities in Kotlin. To that end, the annotations @*NodeEntity* and @*RelationshipEntity* are required, for mapping the nodes and relationships, and the annotation @*Relationship* for adding a connection between them. All three annotations can be observed below in figure 5.4, where snippets of the initial entities are shown.



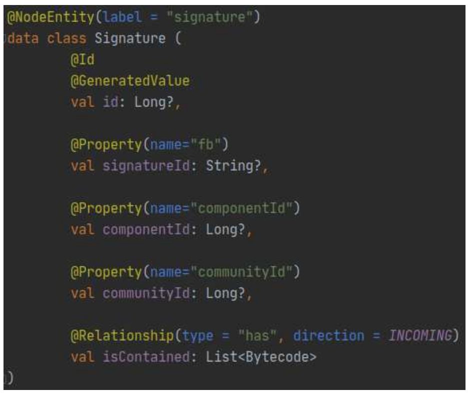
```
@Bean
public org.neo4j.ogm.config.Configuration configuration() {
    org.neo4j.ogm.config.Configuration configuration = new org.neo4j.ogm.config.Configuration.Builder()
            .uri("bolt://neo4j:admin@localhost")
            .build();
    return configuration;
}
```

Figure 5.3: Neo4j configuration



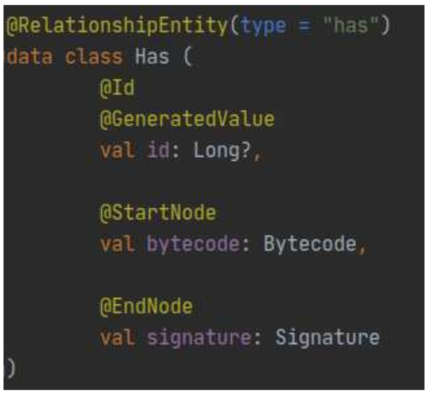
```
@NodeEntity(label = "bytecode")
data class Bytecode(
        @Id
        @GeneratedValue
        val id: Long?,

        @Property(name="id")
        val bytecodeId: Long?,

        @Property(name="componentId")
        val componentId: Long?,

        @Property(name="communityId")
        val communityId: Long?,

        @Relationship(type = "has", direction = OUTGOING)
        val has: List<Signature>
)
```

(a) Smart contract entity



```
@NodeEntity(label = "signature")
data class Signature (
        @Id
        @GeneratedValue
        val id: Long?,

        @Property(name="fb")
        val signatureId: String?,

        @Property(name="componentId")
        val componentId: Long?,

        @Property(name="communityId")
        val communityId: Long?,

        @Relationship(type = "has", direction = INCOMING)
        val isContained: List<Bytecode>
)
```

(b) Function signature entity



```
@RelationshipEntity(type = "has")
data class Has (
        @Id
        @GeneratedValue
        val id: Long?,

        @StartNode
        val bytecode: Bytecode,

        @EndNode
        val signature: Signature
)
```

(c) Relationship entity

Figure 5.4: Code snippets for Object-Graph Mapping (OGM)

Finally, the initial repository is created. Spring Data Neo4j offers two ways of querying resutls in a Neo4j graph database. The first way is using custom queries written in Cypher with a @*Query* annotation since Spring Data Neo4j understands the Cypher query language. That means one can write Cypher queries in a Spring project in the same way as in the Web-Application. As a second way, Spring supports a Domain Specific Language (DSL), from which the framework can build a Cypher query from the method name if it follows certain naming conventions, which are explained further in the Spring Data Neo4j documentation [sprb].

With that, the project set-up is completed. In the following subsection, the actual clustering algorithms and their usage are covered. Moreover, further details are given regarding the timeline and reasons for the choices made.

5.2.2 Applying algortihms

As previously mentioned, similarity algorithms are used in this model to find and observe clusters of smart contracts in the Ethereum ecosystem. To that end, two of the algorithms Neo4j implements are used and experimented with:

- **Jaccard Similarity**

  Jaccard similarity (or Jaccard coefficient), introduced as a term by Paul Jaccard [Jac01], is defined as the size of the intersection divided by the size of the union of two sets, or as

  $$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{5.2}$$

  for two sets A and B. The resulting similarity will always be a number between 0 and 1, with the sets of data being more similar when this number is closer to 1.

  This algorithm works best when the similarity between two things needs to be calculated, or in the case of this research, the similarity between each pair of two smart contracts.

  The clusters produced by this algorithm are expected to be from smart contracts that are "similar" in a traditional sense. Meaning smart contract bytecodes that share the most function signatures will be clustered together.

- **Overlap Similarity**

  Overlap similarity (or Overlap coefficient) is the second algorithm used to create clusters and is defined as the size of the intersection divided by the smaller of the size of the two sets, or

  $$O(A,B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{5.3}$$

  for two sets A and B.

  Similar to Jaccard, this coefficient is also best suited to measure the similarity between two sets of data, and the resulting similarity will always be a number between 0 and 1. However, the best use case for Overlap similarity is when figuring out which things are subsets of others.

That means when contract A contains all function signatures of contract B plus some additional ones, then the results from the two algorithms will differ, or

$$\begin{aligned} &O(A,B) = 1 \\ &but \\ &J(A,B) < 1 \end{aligned} \tag{5.4}$$

for two sets A and B.

After initial experimentation with the similarity algorithms, especially with Jaccard similarity, an interesting observation was made. A large number of smart contracts share the same function signatures. These smart contracts are not equal on a bytecode level since duplicates were already extracted in the preprocessing phase of the research but implement the same interface. Since they are regarded as functionally equal and do not offer any new knowledge to the research, they can be extracted from the dataset.

This procedure was successful since it reduced the number significantly. Afterward, when the community detection algorithms are applied, the resulting subgraphs become of a manageable size.

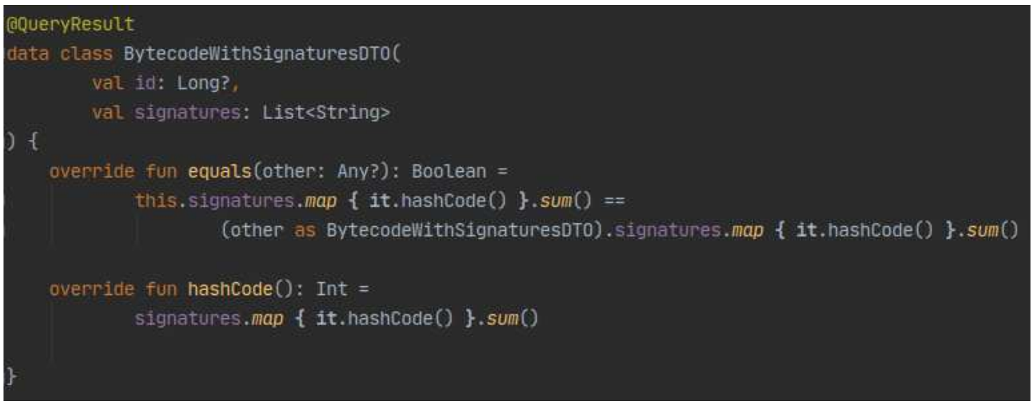
Phase	No. Bytecodes	No. Signatures	No. Relationships
Before removal	235 770	278 103	3 682 573
After removal	74 975	278 103	1 304 623

Table 5.1: Status before and after identical smart contract are removed

This duplicates removal could not be done directly in Neo4j, because to find "identical" smart contracts, Jaccard similarity still needs to be calculated on the whole dataset. Only after, all but one duplicate can be omitted. However, using a Kotlin method takes away a lot of this complexity, and calculating Jaccard is not even necessary. The following steps were taken to remove all duplicates:

- The complete dataset is loaded in memory in objects that has two fields. One, the id property of the smart contract bytecode, and the other a list of all id properties of the function signatures owned by the same contract.
- Both *equals* and *hash* functions are overwritten is such a way that the comparison of an object is based on the list of function signature ids. More specifically, when two of these objects are compared, the system will deem them equal, only if the sum of signatures' hashcodes is equal. In figure 5.5 a code snippet can be observed directly from the project. The annotation @*QueryResult* is added so that Sring understands that the object will be a result of a Cypher query.

- After the whole dataset is in memory, the only thing left is to get all distinct objects, export them to an external CSV file, and load the new version of the dataset in Neo4j in the same way as explained in the previous subsection.



```
@QueryResult
data class BytecodeWithSignaturesDTO(
        val id: Long?,
        val signatures: List<String>
) {
    override fun equals(other: Any?): Boolean =
            this.signatures.map { it.hashCode() }.sum() ==
                    (other as BytecodeWithSignaturesDTO).signatures.map { it.hashCode() }.sum()

    override fun hashCode(): Int =
            signatures.map { it.hashCode() }.sum()

}
```

Figure 5.5: equals and hashCode override

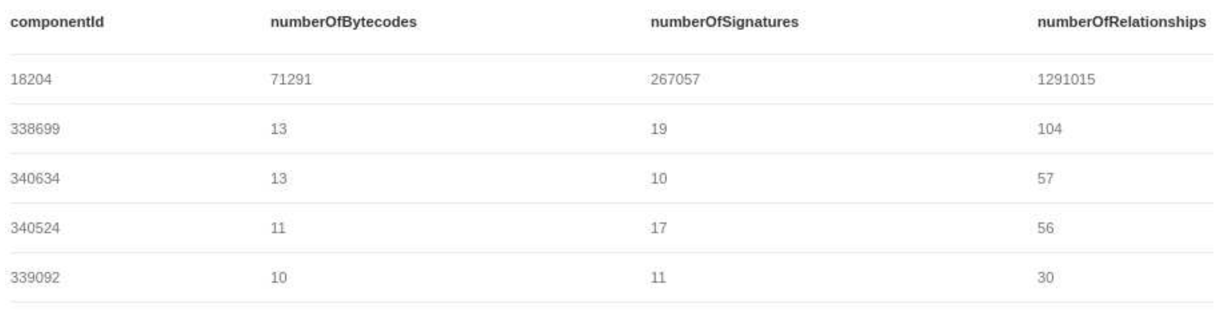
Community Detection

The next step in the implementation process is partitioning the dataset in segments small enough that we can, first of all, work with and calculate the similarity, and second visualize the results in Gephi. To that end, two out of the three community detection algorithms provided by Neo4j were used to partition the graph.

The first algorithm is the Weakly Connected Components. WCC is the only deterministic community detection algorithm that Neo4j implements. It works by finding all sets of connected nodes in a graph and forming subgraphs of the data. When this algorithm is run on the Ethereum dataset persisted in Neo4j, it will create graph partitions of all bytecodes that share signatures. And because smart contracts that do not share any function signatures will not produce clusters of any kind, and consequently are not important to this research, WCC is the correct choice.

The WCC algorithm in Neo4j offers an option to calculate all sets of connected components and assign an integer property to each node. This number represents the id of the connected component to which the node belongs, and for that reason, we name it "componentId".

Below in figure 5.6, a screenshot with information about the dataset in regards to the WCC can be observed, directly taken from the Neo4j Web-Application. The result rows are ordered by the number of smart contracts contained in a partition. The total number of weakly connected components is 3 055.



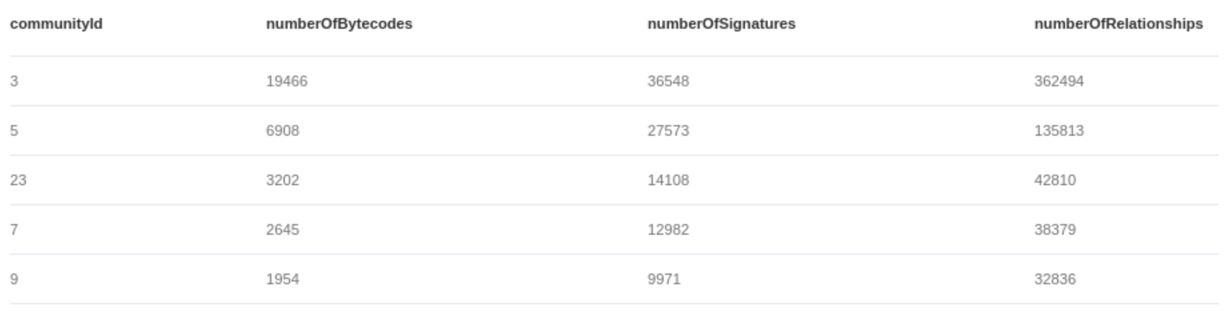
componentId	numberOfBytecodes	numberOfSignatures	numberOfRelationships
18204	71291	267057	1291015
338699	13	19	104
340634	13	10	57
340524	11	17	56
339092	10	11	30

Figure 5.6: First five rows of the dataset in Neo4j ordered by number of smart contract bytecodes per WCC

However, as can be clearly seen, the yielded results of the WCC algorithm are not good enough. Apart from the one significant partition that contains 71 291 smart contract bytecodes, the rest are not usable. That means, the noise around the largest connected component can be filtered out as it does not produce any more knowledge.

The other community detection algorithm used is the Louvain algorithm, introduced by Blondel *et al.* [BGLL08], which finds communities in large-scale networks in a non-deterministic way. Its goal is to maximize a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities [neo15b].

Similar to the Weakly Connected Components algorithm, Louvain offers the option to assign a property to all nodes, as to which community they belong. The name chosen for this property is "communityId". The information concerning the state of the data in Neo4j after running this algorithm and the final state of the dataset in this research can be observed below in figure 5.7, also ordered by the number of smart contracts contained in a Louvain community. Furthermore, the total number of Louvain communities is 3 624.



communityId	numberOfBytecodes	numberOfSignatures	numberOfRelationships
3	19466	36548	362494
5	6908	27573	135813
23	3202	14108	42810
7	2645	12982	38379
9	1954	9971	32836

Figure 5.7: First five rows of the dataset in Neo4j ordered by number of smart contract bytecodes per Louvain community

This algorithm yields numerous communities with more than one hundred smart contracts, and the largest community contains only 19 466, which makes these partitions manageable.

Creating Clusters

Everything that has been done up to this point is the preparation for this final step in the business logic layer, and that is implementing the algorithms that create the actual clusters.

Both similarity algorithms are applied to the dataset, separately for each WCC and Louvain community. For all but the largest WCC and Louvain community, the algorithms can be run directly in Neo4j. For the two remaining partitions, additional logic is applied in the Spring project.

Let us take a partition with an N number of smart contracts as an example. Next, sets of W number smart contracts are taken from the partition, and the algorithm is run on both on W against each other and the remaining bytecodes of N in iterations. With every iteration, the results produced are appended and exported to a CSV file, and the number of bytecodes in N is reduced by W. The state of N is decreased by W after every iteration.

So, for a partition with N number of bytecodes and W number of bytecodes in a window, the number of calculations is

$$\frac{W * (W-1)}{2} + W * (N - W) \tag{5.5}$$

for a single iteration. After each iteration, N becomes $N - W$.

Let us now take the larges WCC as an example with concrete numbers. It contains 71 291 smart contract bytecodes. By the calculation explained in function 4.2, the number of calculations needed for a similarity algorithm to be applied is 2 541 167 695. However, following the logic from above, if W is 100 bytecodes, the calculations for the first iteration will be 7 123 950. Afterward, this number will constantly reduce and the method will increase in speed.

Once the algorithms yield results for a community, the whole set is sent through a method that filters and formats the results, and exports them to CSV files. To better manage the resulting datasets in the presentation layer, results from the algorithms are exported in separate CSV files according to their similarity. More precisely:

- Results with similarities from 0.9 to 1.
- Results with similarities from 0.8 to 0.9.
- Results with similarities from 0.7 to 0.8.
- Results with similarities from 0.6 to 0.7.
- Results with similarities from 0.5 to 0.6.

- Results with similarities from 0.1 to 0.5.

This way, if a dataset for a certain partition is too large, and the local machine does not have to resources to optimally manage it in Gephi, only the data with higher similarities can be loaded. All resulting datasets are exported in separate CSV files.

5.3 Presentation Layer

This final section for this chapter covers this thesis' implementation using the large-scale network visualization tool Gephi. It includes the format of the exported CSV files and the algorithms used for the graph partitions loaded and their configurations.

The file can be imported in different ways, and for this research, two import methods are important.

- **Import CSV file as a Nodes Table**

  When importing the file as a nodes table, Gephi gives all importance to the nodes and their properties. By default, Gephi requires the usage of an id, by which the node can be identified, and an optional label, which can later be used for better visualization. If the imported file has no label column, it will automatically be set as "null". However, its value can later be changed manually.

- **Import CSV file as an Edges Table**

  This type of table, as the name suggests, is used to visualize the data as a network, with nodes and edges connecting them. To that end, two distinct columns are required to be present in the imported CSV file, and those are the "source" and a "target" column. These two columns represent the nodes (or node ids), between which a connection is present. Gephi differentiates a "source" and "target" because it supports directed graphs, but in this thesis that is not important, as we are working with undirected ones. Another feature that is important, is an optional "weight" column can be added to transform the view into a weighted graph. That means that a numerical value is assigned to the edges as "weight" that can be used to measure the distance or strength between the nodes and create clusters for better visualization.

We load data both as nodes and edges. The datasets loaded as a node table contains the smart contract ids, the weakly connected component they belong to, and the Louvain community. Afterward, labels of known contracts will be added for comparing and improving results. For the second type, importing the file as an edges table, the source and target columns represent the ids of the smart contracts, on which the similarity algorithms are run, and the last column is the similarity results as weight.

For each dataset loaded in Gephi, two layout algorithms implemented by Gephi are used for better visualization and evaluation of the clusters. Both algorithms are used to create and visualize the clusters according to the similarities produced by Jaccard and Overlap respectively, which are set as weights in on the edges between all similar nodes. The choice regarding which layout algorithm is chosen depends on the dataset loaded in Gephi, or more precisely on the size and density of the weighted graph.

ForceAtlas2

The first algorithm used to create and visualize the clusters is ForceAtlas2 [JVHB14]. ForceAtlas2 is a force-directed network layout algorithm created especially for Gephi, which simulates a physical system in order to spatialize a network. It works by making the nodes repulse each other and the edges attracting the connected nodes in proportion to the weight, like springs.

Both algorithms are available in Gephi, however, the first version is deemed obsolete. Gephi offers different configurations for the ForceAtlas2, depending on the size and density of the dataset. Some of the more important to visualize better results are:

- **Gravity**

  As the name suggests, the gravity configuration is used to prevent disconnected components to drift too far from each other. This option is a numerical value, and the higher it is, the stronger the gravity will be.

  Gephi also offers another gravity option, called "Strong gravity", which is a check-box option, and when selected, it creates a force that attracts all disconnected components to the center.

- **Scaling**

  This option is a numerical value that when set, configures the size of the view. The larger the scaling value is, the large the graph will be. This option proves useful for this thesis because the datasets loaded differ widely in size.

- **Edge weight**

  This option allows the user to set a numerical value that defines the strength of the weight property of the edges. This option is usually used to better visualize datasets with different sizes.

Gephi's ForceAtlas2 offers other configurations as well, like **LinLog mode**, **Dissuade Hubs**, **Prevent Overlapping**, and more, which are further explained by Jacomy *et al.* [JVHB14].

OpenOrd

The second layout algorithm used for better cluster visualization is Gephi's OpenOrd algorithm [MBKB11]. OpenOrd is a force-directed layout algorithm, designed especially to handle very large graphs. This algorithm is based on the Frutcherman-Reingold algorithm for force-directed layout [FR91].

OpenOrd supports the following configurations:

- **Edge Cut**

  Edge cutting in OpenOrd is specified by a numerical value between 0 and 1. When the edge cut is set to 0, it corresponds to the standard Frutcherman-Reingold layout algorithm with no cutting. When this value is set to a larger number, it produces more clustered results. The default value for edge cutting in OpenOrd is 0.8.

- **Num Threads**

  One disadvantage of the Frutcherman-Reingold algorithm is that it does not scale on large-scale networks. This disadvantage has been since fixed in OpenOrd by allowing the algorithm to run in parallel to increase the speed of computation. Gephi's OpenOrd allows users to configure the number of threads, and each thread will work on a subset of the nodes in the graph. The number of threads available depends on the processing power and resources on the machine. For example, on a quad-core computer, it is recommended to use 3 threads [ope].

- **Num Iterations**

  OpenOrd is not a continuous algorithm like ForceAtlas2, but works with a limited number of iterations set by the user. These iterations are controlled by a simulated annealing type scheduling, and is comprised of the following 5 phases: liquid, expansion, cool-down, crunch, and simmer. This process is explained further by Martin *et al.* [MBKB11].

Similar to ForceAtlas2, this algorithm also supports more configurations, however, for the purposes of this thesis, they were deemed unnecessary.

As previously stated, these two algorithms are used in this thesis for creating and visualizing clusters of Ethereum smart contracts, and the choice depends on the size of the dataset, for both nodes and edges. When dealing with a larger dataset, it is recommended to use OpenOrd, which leaves the smaller ones to ForceAtlas2.

The utilization of Gephi concludes this chapter, and in the next results and analysis of the datasets loaded follow that are expected to answer the research questions of this thesis.

CHAPTER 6

Results and Evaluation

This chapter covers a comprehensive graph evaluation and discussion for the datasets of interfaces of Ethereum smart contracts clustered by the number of function signatures they share, using the two different similarity algorithms, Jaccard and Overlap. The goal of this chapter is to provide results from the model described in the previous two chapters and their evaluation.

To that end, this chapter is organized in the following manner. First, section 6.1 gives an overview of the resulting datasets in Gephi, with information about the clusters produced by both Jaccard and Overlap similarity algorithms. Next, in section 6.2, the set-up of the evaluation process is presented, including labels of known smart contract interfaces taken from related work and used for the evaluation, and the evaluation metrics and parameters, both for selecting exemplary datasets and evaluating the datasets themselves. Afterward, in section 6.3, the actual evaluation and analysis of selected graphs, loaded in Gephi, is conducted. This section is comprised of a qualitative and comparative analysis of selected datasets using the workflow described in section 6.1. Following this, section 6.4 completes this chapter with a discussion about the knowledge gained regarding the clusters created using the two different similarity algorithms and comparing them against labeled interfaces.

It should be noted that a graph analysis is not carried out for the largest weakly connected component, due to the size and density of the $smart\ contract \rightarrow\ function\ signature$ relationships. However, this component has already been partitioned into multiple Louvain communities that are more manageable, in both size and density.

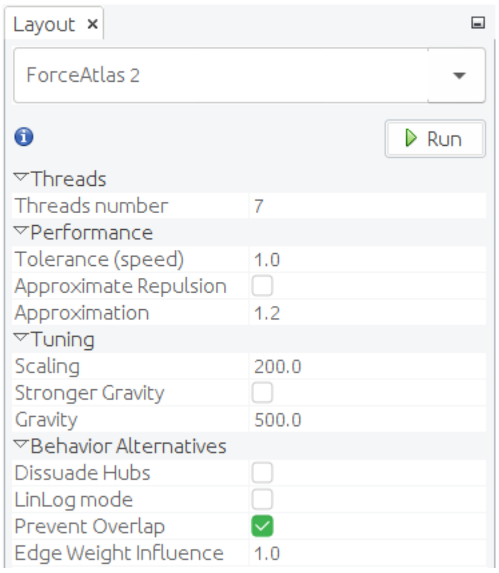
6.1 Results

In table 6.2 and table 6.4 information about the produced datasets from our solution can be observed. The tables contain side-by-side information for datasets produced by running

Jaccard and Overlap similarity algorithms on the 50 largest connected components. The first table covers data produced with a cut-off of 0.5, and the second with 0.1.

Additional information about the structure of the resulting tables can be seen below:

- No. Nodes: Number of Nodes in a dataset.
- No. Edges: Number of Edges produced by the selected clustering algorithms for a dataset.
- No. Clusters: Number of clusters observed for a dataset. The size of a structure that is considered to be a cluster depends on the size of the dataset. All datasets are visualized in Gephi, using the ForceAtlas2 algorithm with the configuration observed in figure 6.1, as it proved to provide the best clusters (depending on the size of the datasets we change the gravity and scale metric).
- Cluster Info: The cluster information is represented as nodes to edges ratios, as comma-separated ratios for each of the observed clusters.
- "/": This sign means that the data loaded in Gephi is too large (too many nodes and/or edges) for it to run layout algorithms without crashing.
- NO: This means that no clusters can be detected in the graph.



Threads	
Threads number	7
Performance	
Tolerance (speed)	1.0
Approximate Repulsion	☐
Approximation	1.2
Tuning	
Scaling	200.0
Stronger Gravity	☐
Gravity	500.0
Behavior Alternatives	
Dissuade Hubs	☐
LinLog mode	☐
Prevent Overlap	✓
Edge Weight Influence	1.0

Figure 6.1: Gephi layout algorithm configuration

No. Nodes	Jaccard			Overlap		
	No. Edges	No. Clusters	Cluster Info	No. Edges	No. Clusters	Cluster Info
19 466	/	/	/	/	/	/
6 908	38 817	3	234/9238, 80/1627, 74/1466	752 264	2	447/31 343, 2 890/441 177
3 202	4 964	3	34/434, 22/212, 14/91	173 797	NO	NO
2 645	22 678	3	170/9 944, 99/3 224, 41/757	364 303	NO	NO
1 954	5 859	3	61/1 422, 72/1 535, 24/275	32 813	1	465/17 580
1 924	55 576	2	288/32 049, 259/22 508	186 456	2	364/54 499, 321/44 821
1 916	3 948	3	54/875, 42/422, 22/120	30 969	NO	NO
1 534	8 012	3	99/2 679, 100/3 141, 78/1 056	48 097	2	430/30 206, 138/8 090
1 172	1 663	2	38/232, 28/247	10 077	2	207/4 044, 48/665
1 055	7 979	2	132/5 767, 87/1 121	22 446	2	159/12 276, 133/4 076
981	1 329	3	21/181, 15/68, 14/90,	10 202	2	206/3 913, 124/3 296
846	1 592	2	57/337, 46/458	14 308	1	183/8 299
820	1 278	3	27/181, 22/122, 11/55	21 932	1	28/7 108
805	1 209	3	38/208, 34/116, 20/173	6 604	1	80/417
705	1 914	2	59/1 162, 22/177	6 744	3	299/2 411 95/1 304, 74/2 309
692	14 035	2	184/11 773, 57/1 445	24 619	2	231/21 434, 68/1 708
689	2 420	3	44/946, 35/588, 29/406	6 255	3	43/803, 39/692, 29/404
670	2 066	2	71/1 225, 30/167	11 959	2	206/5 211, 71/2 306
646	873	2	30/99, 25/165	6 385	1	90/1 831
638	3 054	1	94/2 589	11 808	1	107/5 228
626	1 184	1	41/594	4 828	1	49/1 098
591	779	2	30/183, 22/142	4 637	1	27/299
552	936	1	53/492	3 746	1	394/3 628
487	530	3	27/116, 18/81, 12/44	3 399	2	77/1 720, 44/447
484	664	2	19/151, 17/69	3 391	1	361/3133
480	501	NO	NO	2 123	1	103/1080
471	1 002	2	41/419, 28/103	4 993	1	175/4 231
456	392	NO	NO	3 111	2	80/1 214, 76/1 160
448	391	NO	NO	1 554	1	196/850
433	620	2	22/171, 16/79	1 809	2	36/243, 29/331
425	416	1	16/97	984	2	71/248, 26/144
403	7 516	3	119/6 719, 37/320, 24/135	10 179	2	124/7 352, 64/1 305
393	503	1	31/198	1 545	2	43/164, 32/352
380	414	NO	NO	3 101	1	52/338
370	354	1	14/88	1 193	NO	NO
366	218	NO	NO	780	1	50/155

358	360	NO	NO	2 897	1	243/2 788
357	229	NO	NO	538	1	146/365
347	267	NO	NO	1 433	1	38/443
337	364	1	52/159	1 600	2	82/789, 23/149
323	420	1	29/162	1 795	1	33/418
306	732	2	30/224, 23/250	1 840	2	89/739, 47/713
295	422	1	19/80	1 969	1	91/1 249
294	919	2	36/583, 17/121	2 156	2	41/731, 16/119
269	364	1	15/68	1 061	2	44/157, 36/378
267	168	NO	NO	638	2	24/132, 17/80
258	129	NO	NO	752	2	45/388, 26/177
256	314	1	18/79	1 324	1	182/1 252
251	419	1	20/96	1 228	2	46/468, 41/316
248	5	NO	NO	13 907	NO	NO

Table 6.2: Similarity cut-off 0.5

No. B	Jaccard			Overlap		
	No. S	No. C	Details	No. S	No. C	Details
19 466	/	/	/	/	/	/
6 908	3 779 084	/	/	6 440 974	/	/
3 202	344 717	NO	/	1 237 830	/	/
2 645	1 638 353	/	/	1 422 272	NO	NO
1 954	129 614	NO	NO	329 737	NO	NO
1 924	364 522	2	394/77 076, 274/37 279	907 788	NO	NO
1 916	99 655	3	463/36 756, 156/6 919, 75/2 250	272 769	NO	NO
1 534	211 862	2	590/1239 300, 219/20 608	506 738	NO	NO
1 172	25 598	3	170/10 202, 50/1 017, 35/561	73 637	1	215/21 339
1 055	45 384	2	154/11 642, 101/4 815	145 147	1	469/78 014
981	20 799	3	122/5 582, 102/3 918, 58/1 380	71 414	2	371/45 631, 132/6 326
846	21 329	3	159/12 396, 137/3 884, 77/1 250	75 680	1	165/12 631
820	50 929	2	344/36 975, 96/4 175	152 235	NO	NO
805	16 424	2	104/4 578, 83/2 386	62 078	NO	NO
705	12 684	2	107/2 969, 69/2 337	37 635	2	66/2 080, 54/1 330
692	44 008	2	248/19 662, 66/ 2 127	96 390	1	273/35 872
689	12 017	3	47/985, 44/946, 35/595	70 403	NO	NO
670	16 736	2	220/10 579, 71/2 485	69 019	2	36/630, 35/595
646	12 640	2	96/3 345, 52/1 265	46 602	NO	NO
638	22 977	1	166/13 390	66 311	NO	NO
626	9 504	3	69/2 153, 41/755, 21/224	47 909	1	67/2 175
591	9 733	3	62/1 497, 38/529, 28/351	41 248	NO	NO
552	9 488	2	91/3 412, 49/909	33 818	NO	NO
487	5 833	3	77/2 589, 43/722, 40/618	16 832	1	77/2 925
484	9 001	1	67/1 959	34 132	NO	NO

480	6 284	1	81/2 256	22 167	1	104/4 554
471	10 446	2	37/693, 18/153	39 671	NO	NO
456	8 200	2	98/3 453, 73/1 824	30 945	NO	NO
448	2 504	1	22/203	12 516	NO	NO
433	4 363	2	37/379, 32/478	21 196	2	38/674, 20/180
425	2 712	3	32/434, 29/290, 26/307	9 379	1	30/399
403	11 518	3	124/7 626, 60/1 765, 24/274	31 841	2	124/7 381, 55/1 485
393	4 445	3	36/360, 30/320, 26/368	17 510	1	37/600
380	5 158	2	36/566, 22/214	17 356	1	45/941
370	2 413	2	33/214, 14/91	12 308	1	34/394
366	1 455	1	17/98	6 614	NO	NO
358	4 635	1	60/1 212	20 302	NO	NO
357	1 661	1	49/495	11 193	NO	NO
347	2 027	2	35/587, 14/91	8 254	1	33/528
337	3 428	3	63/1 448, 22/172, 20/171	8 058	1	62/1 479
323	4 555	1	35/573	15 723	1	22/164
306	4 480	3	61/1 647, 33/528, 15/99	13 017	1	77/2 617
295	4 213	3	30/435, 17/136, 17/128	15 088	1	79/3 081
294	4 488	2	38/681, 24/255	13 463	1	23/253
269	3 532	3	61/1 342, 32/490, 26/281	8 778	1	51/949
267	1 856	1	18/161	9 139	NO	NO
258	1 630	1	35/537, 21/195	7 275	1	50/1 026
256	4 244	1	21/210	9 471	NO	NO
251	2 249	2	45/821 42/565	6 018	2	47/809, 42/680
248	30 384	NO	NO	30 384	NO	NO

Table 6.4: Similarity cut-off 0.1

6.2 Evaluation Setup

In order to properly conduct our evaluation, two steps are required. The first one is adding knowledge to the datasets, in the form of labels for known smart contract interfaces, and the second is defining the correct evaluation metrics. Both steps are described in the following subsections.

6.2.1 Labels

This section covers the information concerning the labels of known smart contracts. These labels are loaded in Gephi in the later stages of this thesis to accurately evaluate the validity of the clusters produced by both similarity algorithms.

It is important to note that finding and extracting labels of know smart contracts are not part of this thesis, but are provided by TU Wien, as explained in [dAS19]. These labels were later mapped to our existing dataset by the set of ABI function signatures, which at this phase of the research, are unique for each smart contract. Two sets of labels are provided for the purposes of this thesis, the main label, and a sub-label. Each smart contract in the dataset, that contains a label, also contains a sub-label.

Information about the current state of the dataset, regarding the labels and sub-labels, can be observed in table 6.5, which contains the list of main labels, the number of smart contract interfaces in which this label can be found, and the number of Neo4j connected components containing a smart contract with that label.

Label	No. Interfaces	No. Communities
No Label	51 250	/
ercToken	19 345	141
Token	3 893	91
WalletFactory	216	65
Wallet	207	23
Mayfly2	30	20
Controller	20	1
Ambi2	8	3
dDoS2016	3	3
DefaultSweeper	2	1
Gastoken	1	1

Table 6.5: Information about labels of known smart contract interfaces

As can be seen, some of the labels appear in smart contracts that are located in a larger number of communities. To reduce that number, we make use of the sub-labels. The list of all sub-labels for each of the main ones can be observed below, in table 6.6.

Label	List of sub-labels
ercToken	erc20, erc1462, erc721, erc777, erc1155, erc1644
Token	Token
WalletFactory	WalletFactory
Wallet	multisig WalletSimple/BitGo, smart GnosisSafe, intermediatewallet, multisig Christian Lundkvist, multisig Stefan George, consumer wallet, multisig WalletSimple/BitGo forwarder, timelocked wallet, autowallet, basicwallet, controlled, dapper, eidoo wallet, ether wallet 1, ether wallet 2, logicproxywallet, multisig Gavin Wood/Ethereum/Parity, multisig Ivt, multisig Julien Niset/Argent, multisig NiftyWallet, multisig Teambrella Wallet, multisig Unchained Capital, simple wallet, simple wallet 2, simple wallet 3, smart Julien Niset/Argent, smartwallet, spendable wallet, wallet1
Mayfly2	Mayfly2
Controller	Controller
Ambi2	Ambi2
dDoS2016	dDoS2016
DefaultSweeper	DefaultSweeper
Gastoken	Gastoken

Table 6.6: List of sub-labels for each of the labels of known smart contract interfaces

For two main labels, "ectToken" and "Wallet", the sets of sub-labels produce results and for the most part reduce the number of components, in which they can be found. This is especially noticeable with the "Wallet" label. Out of this set of sub-labels, only three can be found in smart contracts located in more than one community. "multisig WalletSimple/BitGo" can be found in three, and "smart GnosisSafe" and "intermediatewallet" can be found in two. All others only exist in smart contracts contained in a single community.

6.2.2 Evaluation Metrics and Parameters

This section consists of two separate evaluation parameters used for two different parts of the evaluation process, and they are **evaluation parameters for selecting a correct dataset** and **evaluation parameters for the produced resuts**.

Evaluation parameters for selecting a correct dataset

As it was already discussed, loading the whole dataset in the visualization tool is infeasible. Even when the data is partitioned, and loaded separately, evaluation of all 3 000 plus components would be too time and space consuming for this research. That is why partitions that are believed to produce the best results should be selected. This choice depends on the following parameters:

- **Size of the dataset**

  The first parameter for dataset selection is the size of the dataset. The number of smart contract interfaces of the chosen dataset should not be too large as to overload Gephi, considering the resource capacity of the available machine, and at the same time not too small, so it can produce reliable results. For that reason, WCC and Louvain components with more than 2 000, and less than 1 000 smart contracts are not considered as candidates for the evaluation phase.

- **Number and size of clusters in the dataset**

  It is obvious that clusters play a significant role in this thesis. However, because the algorithms that produce the clusters are run separately on all components, they might not produce a cluster for a certain community. A dataset is considered as a candidate for further evaluation if it contains at least one large visible cluster, produced by at least one of the two similarity algorithms (Jaccard and Overlap). Furthermore, a large cluster is considered any structure in the graph that contains at least 50 nodes, 1 000 edges, and an average edge weight of 0.6 (the edge weight is represented by the similarity between the source and target node).

- **Number of known smart contracts in the dataset**

  It can be observed in the previous subsection that labeled smart contracts can be found in numerous components, and they are of significant importance to the evaluation phase. For that reason, datasets that contain a larger amount of known smart contracts are considered as better candidates for evaluation. Preferably the labeled interfaces are located in a cluster for better results.

- **Diversity of known smart contracts across the datasets**

  Labels and sub-labels of the same type can be found in numerous connected components, as can be seen in table 6.5. As candidate datasets for evaluation are considered those, in which as much known smart contracts with a certain sub-label as possible are contained. Meaning that, for example, if all known smart contracts with a certain label or sub-label are located in a single component, that one is more likely to be selected.

Evaluation parameters for the produced resuts

The other type of parameters is defined to evaluate the validity of the resulting graphs. Once a dataset is loaded in Gephy and layout algorithms are run to visualize the clusters of the weighted graph, the following metrics are implemented for evaluation:

- **Cluster quality**

  The quality of a resulting cluster depends highly on the similarity between its nodes. Depending on the dataset, clusters differ in regards to quality, even if they look similar to the user. For example, a cluster with an average similarity of 0.4 between edges can look similar to a one with 0.8. The higher the average similarity is between the edges of a single cluster, the more the validity of the results increases.

Another metric to better evaluate the clusters is the density of the cluster or the ratio of nodes to edges in a single cluster.

- **Difference of results with different similarity algorithms**

  Since two different similarity algorithms are applied to each dataset, the resulting clusters may differ, both in size and density. If the difference between the resulting clusters remains small, it will increase the validity of the results.

- **Evaluation using labels of known smart contracts**

  Since we have chosen datasets with a large number of labeled smart contract interfaces, it is expected that those interfaces are contained within a cluster. To that end, two different ratios are of interest to this research, in regards to the labels of known smart contracts in a dataset, and they are defined by the following two functions. The first one is the ratio of labeled to unlabeled smart contracts in a single cluster, and the second is the ratio of labeled smart contracts inside a cluster to labeled smart contracts outside one. For both evaluation parameters, a larger ratio on the side on labeled smart contracts are expected for better results.

6.3 Graph Analysis of Connected Components

Following the evaluation metrics for dataset selection in the previous section, three datasets that fit the criteria were chosen for the next phase, covered in the following subsections.

- Basic information about the dataset.
- Evaluation of clusters for Jaccard similarity algorithm.
- Evaluation of clusters for Overlap similarity algorithm.
- Comparative analysis for the two different clustering methods.

All following datasets are visualized in Gephi, using the ForceAtlas2 algorithm with the configuration observed in figure 6.1, with adjustments in the gravity and scale metrics, depending on the size and density of the data.

6.3.1 Dataset One

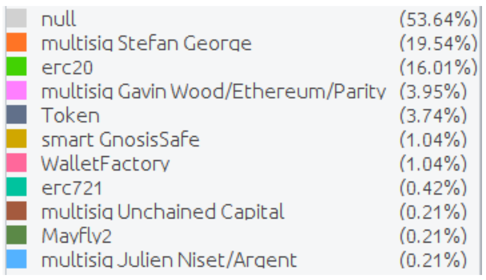
Basic Information

Basic infromation about the first selected dataset can be observed in table 6.7.

Name	Value
Number of interfaces	1 055
Number of function signatures	5 361
Number of labeled interfaces	223
Labels occurring	Wallet, WalletFactory, ercToken, Token, Mavfly2
Most frequent label	Wallet
Most frequent sub-label	multisig Stefan George

Table 6.7: Basic information for dataset one

The color chart for this dataset in regards to the labels (or sub-labels) of known smart contracts can be observed in figure 6.2.



⬜	null	(53.64%)
🟧	multisig Stefan George	(19.54%)
🟩	erc20	(16.01%)
🟪	multisig Gavin Wood/Ethereum/Parity	(3.95%)
⬛	Token	(3.74%)
🟨	smart GnosisSafe	(1.04%)
🟥	WalletFactory	(1.04%)
🟩	erc721	(0.42%)
🟫	multisig Unchained Capital	(0.21%)
🟩	Mayfly2	(0.21%)
🟦	multisig Julien Niset/Argent	(0.21%)

Figure 6.2: Color chart for dataset one

Evaluation of clusters for Jaccard similarity

In figure 6.3 the resulting graph can be observed for the first dataset produced by running the ForceAtlas2 layout algorithm.

In the figure, two clusters of a reasonable size can be observed, and numerous small ones, with a maximum of ten nodes. The first large cluster is the one in the center of the figure, containing the majority of known smart contract interfaces with label "Wallet" and sub-label "multisig Stefan George" (orange color according to the color chart defined above). It contains a total of 132 nodes, and 5 767 edges, and the average edge weight of all edges in the cluster are 0.67.

Out of all nodes in the cluster, 82 have the label "Wallet" and sub-label "multisig Stefan George" (62.88%), 50 interfaces are unknown (35.61%), and 2 have the label "Token" (1.52%). That means, that the ratio of "Wallet" interfaces to unknown interfaces inside a cluster is 41 : 26.

Outside of the cluster, 38 nodes have the same label as the majority of the nodes inside, which makes the ratio of "Wallet" interfaces inside a single cluster to those outside the same 41 : 19. However, if we use the sub-labels for this equation, the ratio becomes 82 : 11.

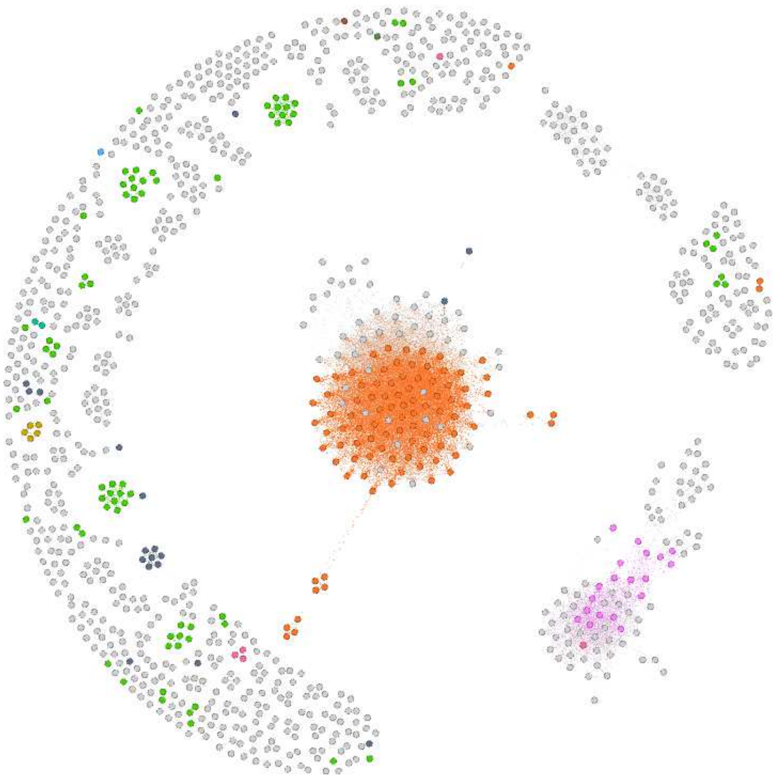


Figure 6.3: Graph produced by Jaccard similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset one

Just by looking at the other large cluster, again containing nodes with the label "Wallet", but sub-label "multisig Gavin Wood/Ethereum/Parity" (purple color), we can expect worse results than the previous. This ratio of nodes to edges of the cluster is 87 : 1 121, and the average edge weight of the cluster are 0.65. From the nodes, 19 have the label "Wallet" and sub-label "multisig Gavin Wood/Ethereum/Parity" (21.84%), 1 has the label "Wallet Factory" (1.15%), and 67 are unknown (77.01%). The ratio of "Wallet "interfaces to the others inside the cluster is 19 : 68.

However, these 19 nodes represent the total number known interfaces with a sub-label "multisig Gavin Wood/Ethereum/Parity", not just in this dataset, but in the whole Ethereum dataset of distinct interfaces.

The rest of the small clusters, from 5 to 15 nodes, contain known interfaces with different labels, such as "ercToken", "Token", and "Wallet", or green, dark gray, and brown respectively. An interesting observation, according to the figure, is that known smart contract interfaces with the label "ercToken" and sub-label "erc20" (green color), can be found in several small clusters. However, when the similarity cut-off is reduced to 0.1, they converge to one single cluster. Other observations are not made when the similarity cut-off is changed.

Evaluation of clusters for Overlap similarity

For this dataset, Overlap similarity with a cut-off at 0.5, finds a lot more edges than Jaccard, or 22 446 to Jaccard's 7 979, as it can be obesrved in figure 6.4.

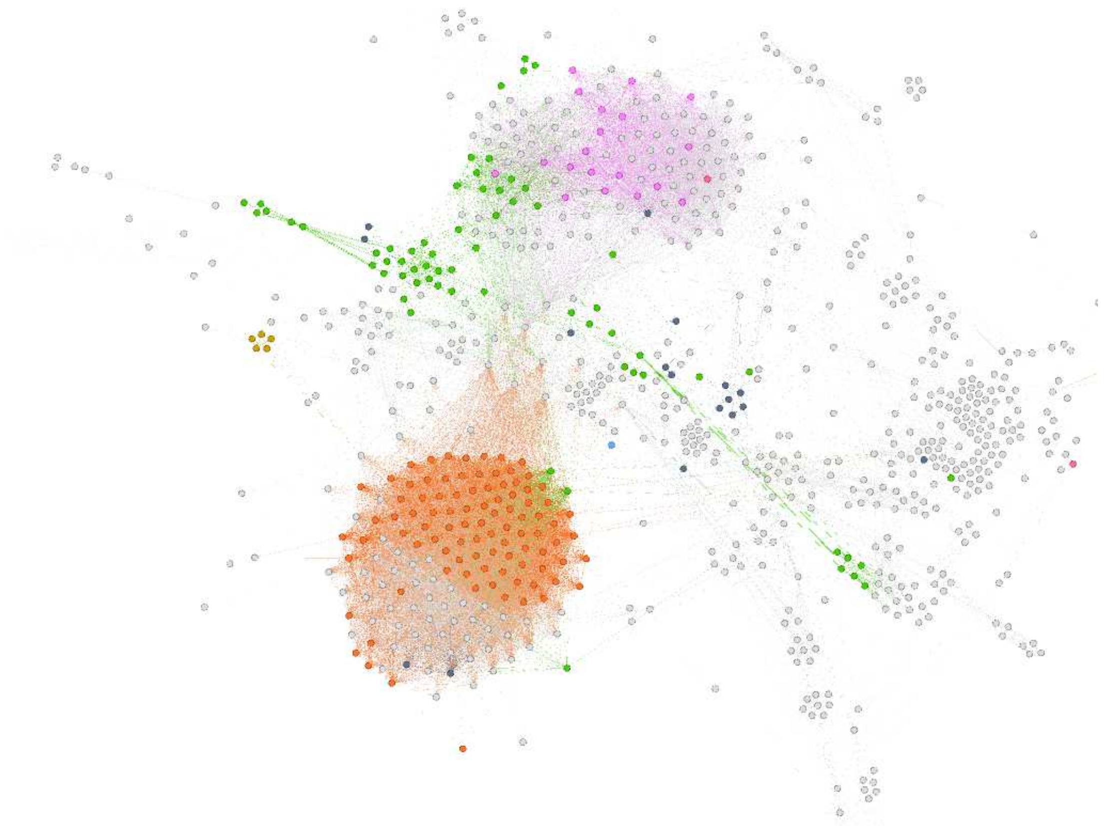


Figure 6.4: Graph produced by Overlap similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset one

Similar to Jaccard, the Overlap algorithm also produces two large clusters (orange and purple nodes), however as it can be observed in the figure, due to the larger amount of edges, the data looks less organized. The larger one of the two clusters has a nodes to edges ratio of 159 : 12 276, and an average edge weight of 0.8.

The number of unknown interfaces in the cluster is 60 (37.75%). From the labeled ones 3 have a label "ercToken" (1.89%), 2 "Token" (1.26%), and all other 94 have the label "Wallet" and sub-label "multisig Stefan George" (59.12%). The ratio of "Wallet" interfaces to the other in the cluster is 94 : 65, and all of them are contained in this cluster.

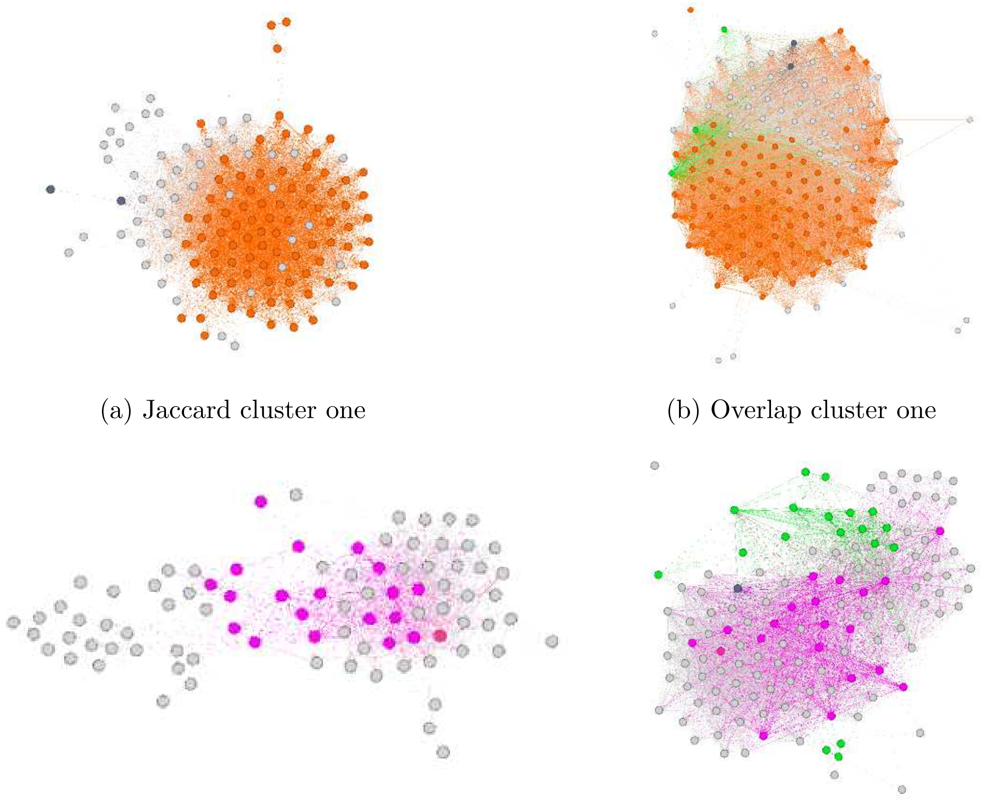
The other large cluster produced by the Overlap algorithm has a nodes to edges ratio of 133 : 4 076, and an average edge weight of 0.69. The number of interfaces labeled with "Wallet", "ercToken", "WalletFactiory", and "Token" are 19 (14.29%), 18 (13.53%), 1 (0.75%) and 1 (0.75%) respectively. The number of unknown interfaces is 96 (70.68%). An interesting phenomenon in this cluster is that a similar number of known smart contract interfaces of two different types can be found, interfaces with label "Wallet" and sub-label "multisig Gavin Wood/Ethereum/Parity", and those with label "ercToken" and sub-label "erc20". For that reason, ratios are calculated for both types of interfaces.

The ratios of known to unknown interfaces in a single cluster are 19 : 114 for "Wallet" interfaces and 18 : 115 for "ercToken" interfaces. Finally, the ratios for known interfaces of a certain type inside and outside a single cluster is 18 : 62 for "ercToken" interfaces, and the interfaces with sub-label "multisig Gavin Wood/Ethereum/Parity" are all contained within this cluster. As mentioned before, due to the nodes to edges ratio in the dataset the graph is scattered, so finding other clusters is difficult.

Comparative analysis of clustering approaches

Several observations can be made when comparing the produced clusters using the two different approaches. When comparing the two clustering algorithms for the dataset, the first thing that can be noticed is the difference in the nodes to edges ratio for the whole dataset. The dataset produced using Overlap similarity with a cut-off 0.5 (the same cut-off is used for both algorithms) has a significantly larger number of edges, or 22 446 to Jaccard's 7 979, which means that a lot of the smart contract interfaces in this clusters are subsets of others when it comes to the function signatures they share.

In figure 6.5 side by side comparisons of the two largest clusters produced by both similarity algorithms can be observed. It can be noticed from the figures that the clusters produced by the Overlap algorithm are larger and denser. When looking at the clusters without having any knowledge of labeled interfaces, overlap produces the better clusters, with average edge weights of 0.8 for the first and 0.69 for the second cluster. However, when labels and sub-labels are introduced, it can be noticed that Overlap finds similarities between "Wallet" and "ercToken" smart contract interfaces, which is especially obvious in the second cluster.


(a) Jaccard cluster one (b) Overlap cluster one

(c) Jaccard cluster two (d) Overlap cluster two

Figure 6.5: Side by side comparison of clusters for dataset one

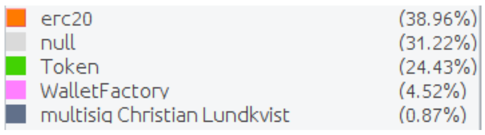
6.3.2 Dataset Two

Basic Information

Information for the next dataset evaluated in Gephi can be observed in table 6.8, and below, on figure 6.6, the color chart of the dataset is presented. This dataset is the largest one of those selected, in both the number of known smart contract interfaces and in total. It represents the sixth-largest Louvain community.

Name	Value
Number of interfaces	1 924
Number of function signatures	6 829
Number of labeled interfaces	791
Labels occurring	Wallet, WalletFactory, ercToken, Token
Most frequent label	ercToken
Most frequent sub-label	erc20

Table 6.8: Basic information for dataset two



🟧	erc20	(38.96%)
⬜	null	(31.22%)
🟩	Token	(24.43%)
🟪	WalletFactory	(4.52%)
⬛	multisig Christian Lundkvist	(0.87%)

Figure 6.6: Color chart for dataset two

Evaluation of clusters for Jaccard similarity

In figure 6.7 the resulting graph can be observed for the first dataset produced by running the ForceAtlas2 layout algorithm.

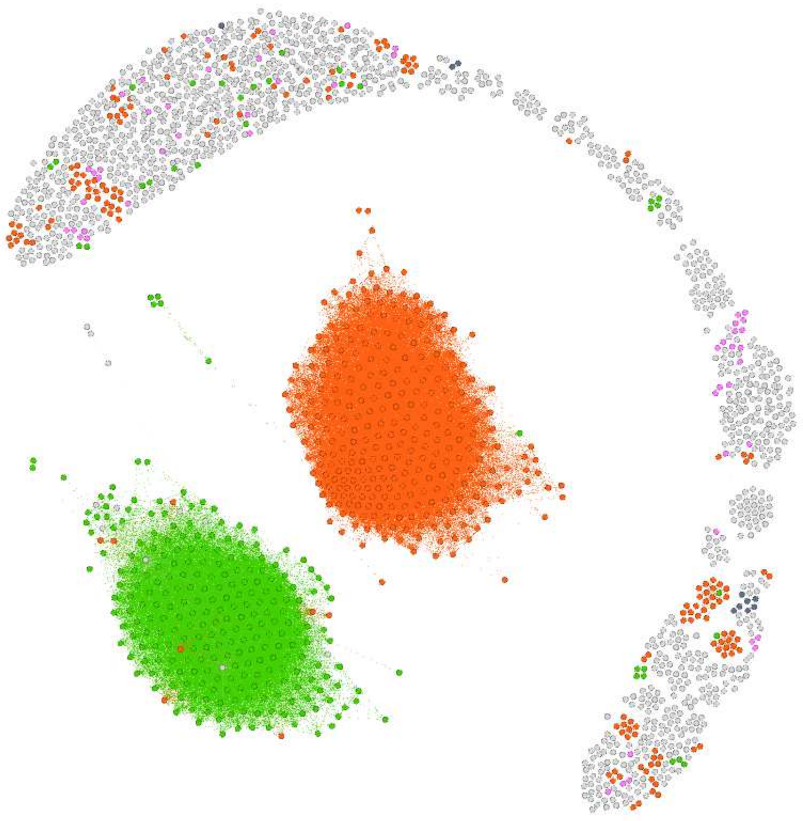


Figure 6.7: Graph produced by Jaccard similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset two

Similar to the previous dataset loaded in Gephi, two large clusters can be observed in the figure, as well as numerous small ones. However, while the clusters in the dataset one were both comprised of "Wallet" interfaces, in this one the clusters contain known interfaces of two different types, namely "Token" and "ercToken". Another "first glance" observation from the figure is that these clusters are very pure, in a sense that hardly any unknown interfaces can be found. Meaning, it is safe to say that the possibility of belonging to the same type of smart contracts is high.

The first large cluster, containing the majority of "ercToken" interfaces (orange color according to the coloring scheme defined above), is almost completely a pure cluster with no unknown interfaces, 1 labeled with "Token", and 287 "ercToken", for a total of 288 nodes, which is 99.65%. The ratio of nodes to edges is 288 : 31 049 and the average edge weight of the cluster is 0.66.

The ratio of "ercToken" interfaces inside the cluster to those outside is 287 : 161, which means that a lot of smart contracts of this type are scattered outside the cluster, some grouped in small clusters to 10 nodes, and some on their own.

On the other hand, the second cluster is comprised mostly of smart contract interfaces that have the label "Token" (green color), or more precisely 243 "Token" interfaces, 8 "ercToken", and 8 unknown. The total size of this cluster is 259 nodes to 22 508 edges, and its edge weight is 0.66.

The ratio of "Token" interfaces to others inside the cluster is 243 : 16, and the ratio of these inside the cluster to that ouside is 243 : 38.

Evaluation of clusters for Overlap similarity

The graph produced by Overlap similarity with a cut-off at 0.5 can be obesrved in figure 6.8.

As was the case with the previous dataset, Overlap calculates more similarities, which results in a graph with larger and denser clusters, but also not as clearly separated from the rest of the data.

The first of the two large clusters, containing the majority of "ercToken" interfaces (orange color), has a nodes to edges ratio 364 : 54 499, and an average edge weight of 0.75. From the nodes, 334 are labeled "ercToken" (91.76%), 6 "Token" (1.65%), 5 "WalletFactory" (1.37%), and 19 are unlabeled (5.22%), which makes the ratio of "ercToken" interfaces to other in the cluster 334 : 20. However, the ratio of "ercToken" interfaces inside to those outside of the cluster is 334 : 104 or 167 : 57.

The other large cluster, containing the majority of "Token" interfaces (green color), has a total of 321 nodes, from which 263 are labeled "Token" (81.93%), 13 "ercToken" (4.05%), and 45 are unlabeled (14.02%). The cluster also has 44 821 edges, and an average edge weight of 0.76.

The ratio of "Token" interfaces to others in the cluster is 263 : 58 and the ratio of "Token" interfaces inside to those outside the cluster is 263 : 18.

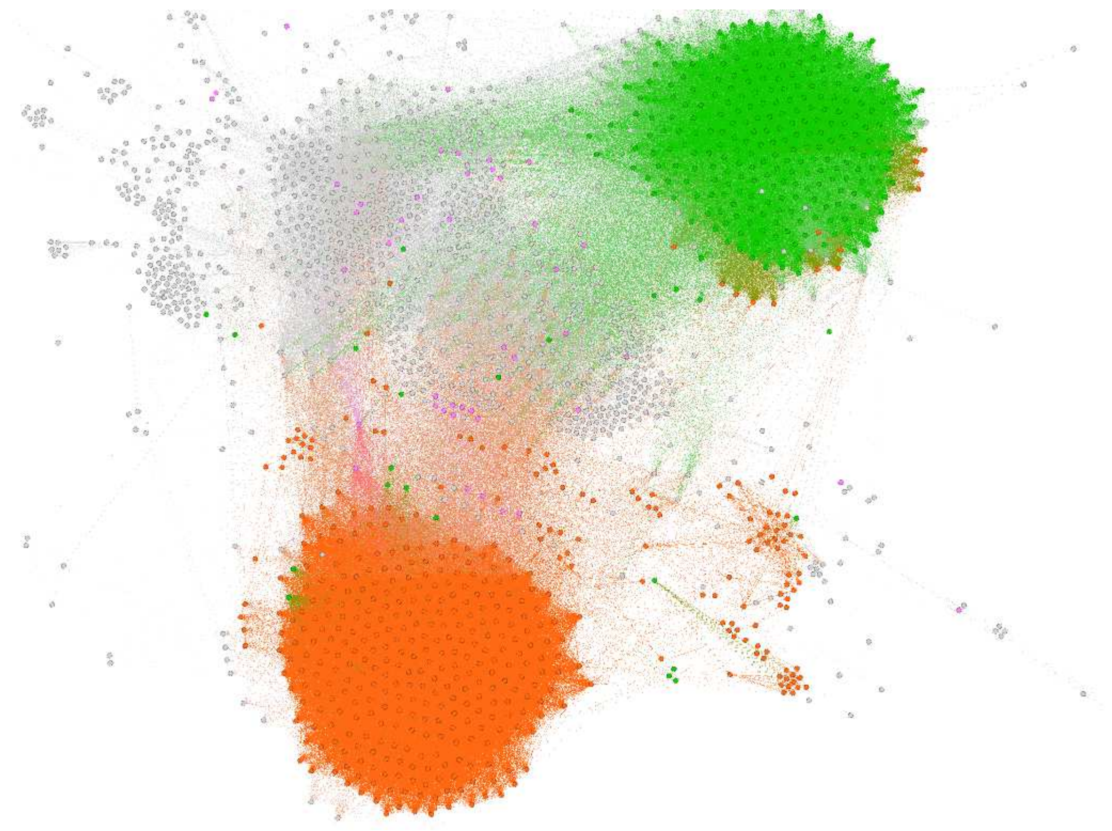


Figure 6.8: Graph produced by Overlap similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset two

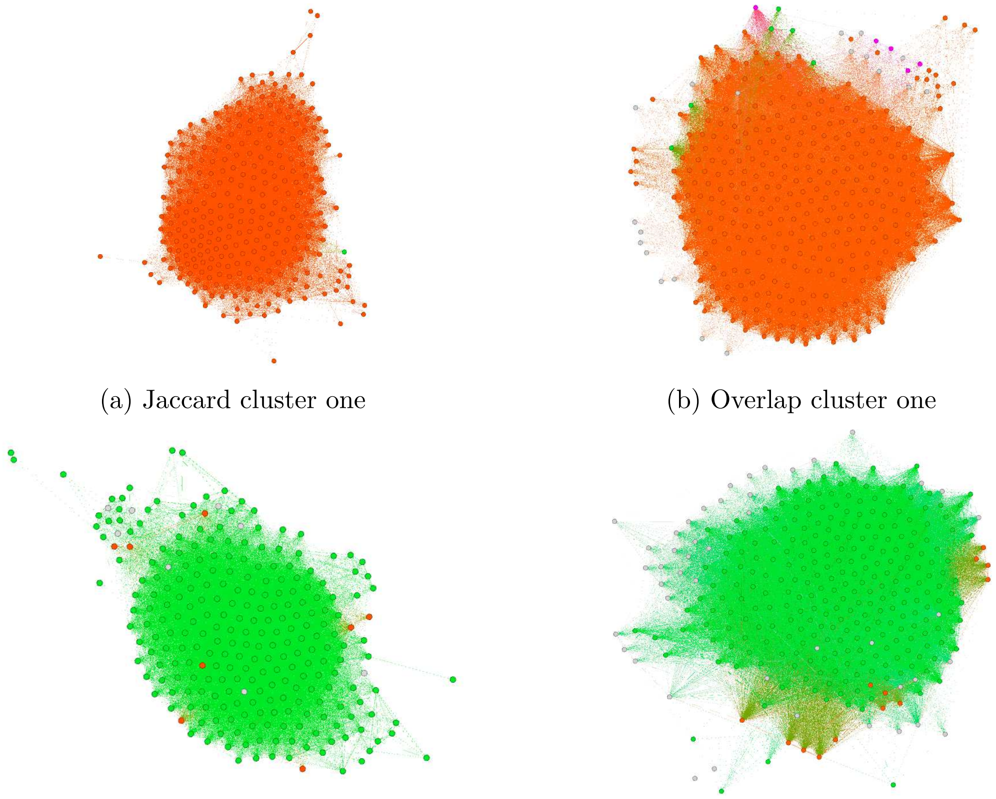
Comparative analysis of clusters

Figure 6.9 shows a side-by-side comparison of the two large clusters produced by Jaccard and Overlap similaritaty respectively.

As was the case with the previous dataset, Overlap again produces larger and denser clusters with higher average edge weights for both, as well as a denser graph for the whole dataset.

That means, that for this dataset, the Overlap algorithm finds more similarities than Jaccard. For example, when observing the first set of clusters in figure 6.9 (*a* and *b*), the one produced by the Overlap algorithm has more "ercToken" interfaces than the one produced by the Jaccard algorithm, more precisely 334 and 241 for Overlap and Jaccard respectively. Thus, a large number of "ercToken" interfaces, which are scattered in the graph produced by running the Jaccard algorithm, are converging to a single cluster in the one produced by Overlap. The situation is the same as the one with the other cluster, *c* and *d* in the figure.

However, it should be stated that even though the clusters produced by Overlap similarity



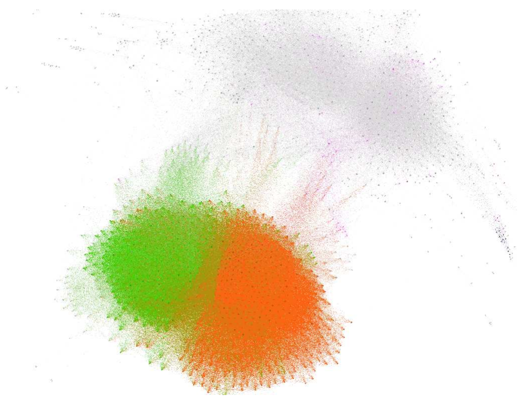
(a) Jaccard cluster one (b) Overlap cluster one

(c) Jaccard cluster two (d) Overlap cluster two

Figure 6.9: Side by side comparison of clusters for dataset two

are larger in both nodes and edges, in Jaccard's clusters a clear separation between them can be observed. Both of Overlap's clusters have a larger number of "false" nodes, meaning that they contain known smart contract interfaces with different labels than the majority, and that is especially noticeable with the cluster in figure 6.10 (b), which has 5 "WalletFactory" interfaces and 6 "Token" interfaces, as well as more unknown ones compared to the one produced by Jaccard.

This cluster separation can be especially observed when the cut-off for Jaccard and Overlap is reduced to 0.1. In figure 6.10, the comparison between the two produced datasets in Gephi can be observed. When looking at the graph produced by Jaccard, in figure 6.10 (a), even though there are some similarities between "ercToken" and "Token" can be observed as they are pulled close to each other, a clear separation between the two types of known interfaces and the unknown ones can be seen. On the other hand, when looking at the graph produced by Overlap in figure 6.10 (b), these separations cannot be observed, which means that for this dataset, Jaccard distinguishes interfaces better when based on their function signatures.


(a) Jaccard

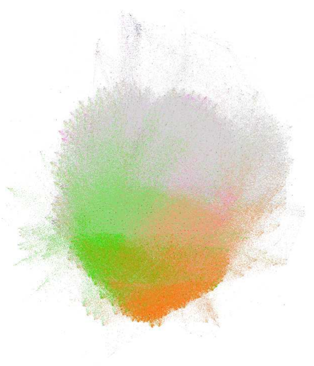

(b) Overlap

Figure 6.10: Comparison of Jaccard and Overlap graphs when the cut-off is set to 0.1

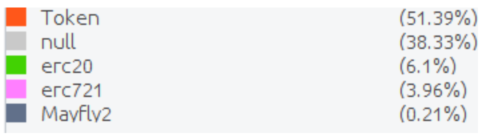
6.3.3 Dataset Three

Basic Information

The basic information for the third and last dataset examined can be observed in table 6.9, and below, in figure 6.11, the color chart for the graph loaded in Gephi.

Name	Value
Number of interfaces	1 534
Number of function signatures	11 910
Number of labeled interfaces	576
Labels occurring	ercToken, Token, Mayfly2
Most frequent label	Token
Most frequent sub-label	Token

Table 6.9: Basic information for dataset three



🟧	Token	(51.39%)
⬜	null	(38.33%)
🟩	erc20	(6.1%)
🟪	erc721	(3.96%)
⬛	Mayfly2	(0.21%)

Figure 6.11: Color chart for dataset three

Evaluation of clusters for Jaccard similarity

In figure 6.12 the Jaccard graph for dataset three loaded in Gephi can be observed. The Jaccard similarity algorithm was run with 0.5 cut-off and the layout algorithm used to visualize the graph was ForceAtlas2.

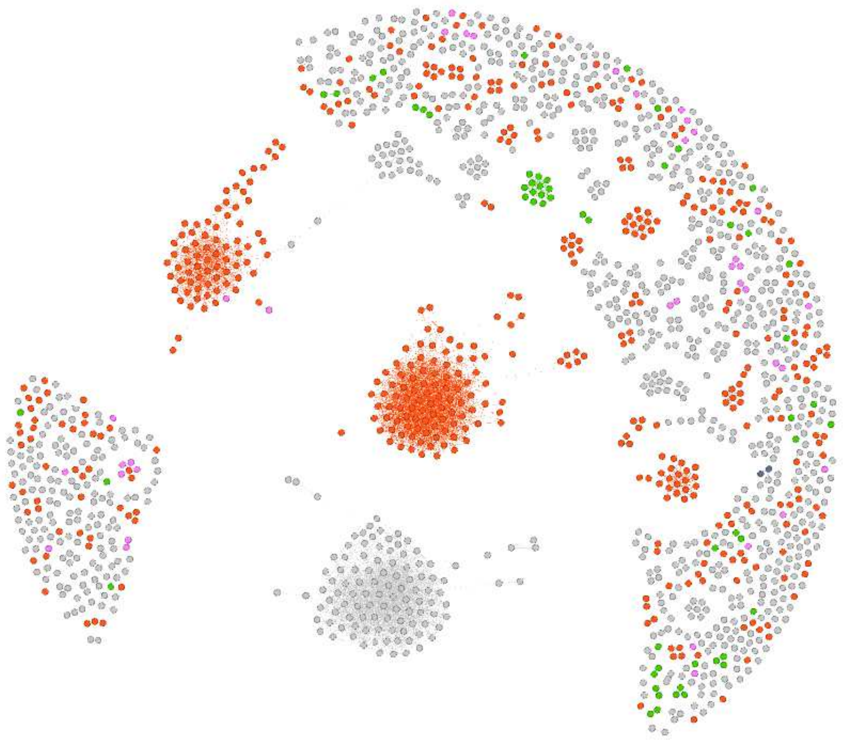


Figure 6.12: Graph produced by Jaccard similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset three

Three distinct large clusters can be observed in the figure. Unlike the graphs produced from the two previous datasets, this one contains a clearly defined cluster without known smart contract interfaces. This cluster is densely connected with a node to edges ratio of 99 : 2 679, and an average edge weight of 0.61.

The other two clusters are comprised almost exclusively of "Token" interfaces. The larger of the two, the one located in the center of the figure, contains only "Token" interfaces, or more precisely 100 "Token" interfaces and 3 141 edges between them. The average edge weight of this cluster is 0.61.

The situation is similar to the other cluster as well, which contains nodes to edges ratio of 78 : 1 056, and an average edge weight of 0.61. From the nodes 74 are "Token" interfaces, 2 are "ercToken", and 2 are unlabeled.

Since these two clusters both have a majority of known smart contract interfaces of the same type but are still evaluated as separate clusters, the ratio of "Token" interfaces inside to those outside is effected negatively. Thus, this ratio for the larger cluster of the two is 100 : 380, and for the other one 78 : 402. Other smaller clusters can also be noticed in the figure, up to 20 nodes, by nodes labeled with "Token" and "ercToken", however, due to their size, they will not be evaluated separately.

When the cut-off of the Jaccard similarity algorithm is reduced to 0.1 more insight can be gained for this thesis, with the most significant one being that all nodes labeled as "Token" and "ercToken" interfaces cluster together in one large cluster. This phenomenon can be observed in figure 6.13, where we can notice that the data is separated into two large partitions, one containing all "Token" and "ercToken" interfaces, and the other the majority of unlabeled ones. This is further proof of the similarity between "Token" and "ercToken" interfaces, which can also be observed in the Jaccard produced graph with cut-off 0.1 on the previous dataset, in figure 6.10 (a). However, the average edge weight of the labeled cluster is 0.2, which for an algorithm that calculates similarity based on shared function signatures of smart contracts is very low.

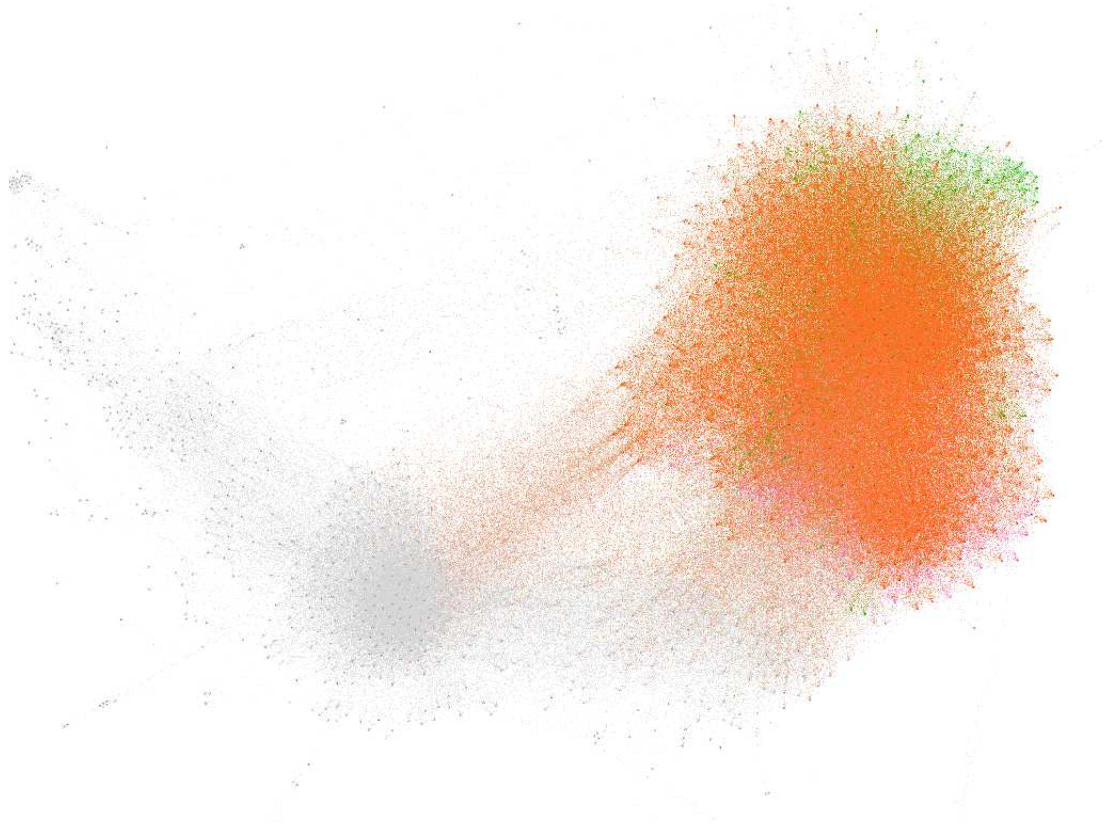


Figure 6.13: Graph produced by Jaccard similarity with cut-off 0.1, visualized in Gephi using ForceAtlas2 for dataset three

Evaluation of clusters for Overlap similarity

In figure 6.14, the resulting graph produced by Overlap similarity with a cut-off at 0.5 can be observed. Same as before, the used layout algorithm for cluster visualization is ForceAtlas2.

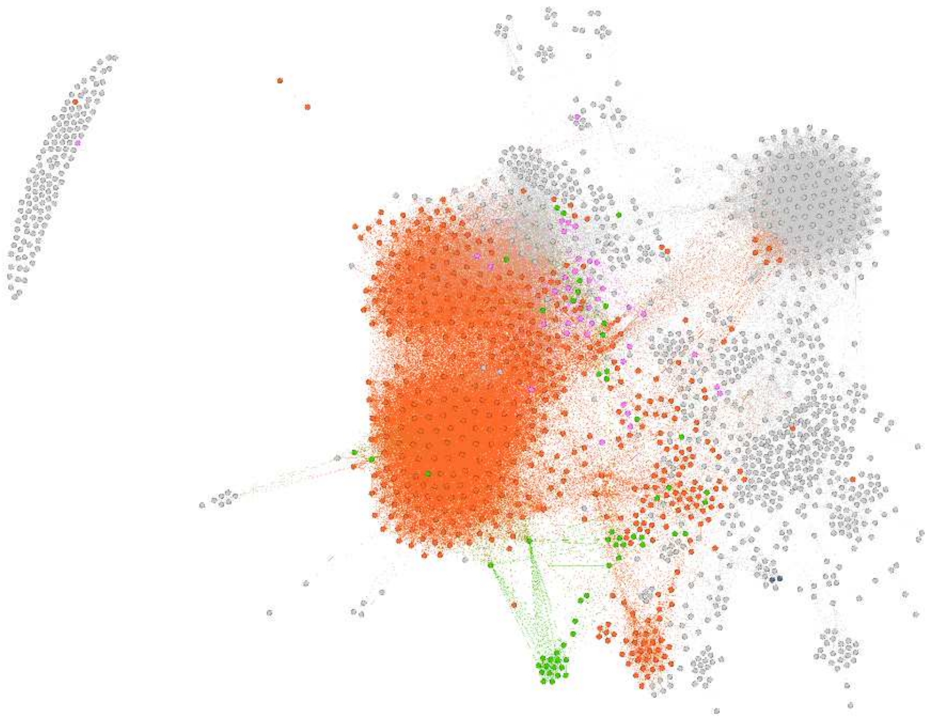


Figure 6.14: Graph produced by Overlap similarity with cut-off 0.5, visualized in Gephi using ForceAtlas2 for dataset three

As expected, the Overlap similarity algorithm produced a denser, more clustered graph, and found more similarities than Jaccard. Two clusters are of interest in this graph. The first one is the large partition of clustered data in the center of the figure, containing a majority of "Token" interfaces. The node to edges ratio of this cluster is 430 : 30 206, and its average edge weight are 0.62. From all nodes, 327 are "Token" interfaces (76.05%), 60 "ercToken", and 43 are unlabeled. Moreover, two different subtypes of "ercToken" smart contract interfaces can be found in the cluster, more specifically 25 interfaces with the sub-label "erc721" (4.19%) and 18 with the sub-label "erc20" (4.19%). They are represented by the purple and green colored nodes respectively. The ratio of "Token" interfaces in the cluster to other ones is 327 : 103, and the ratio of "Token" interfaces inside the cluster to those outside the same one is 327 : 153.

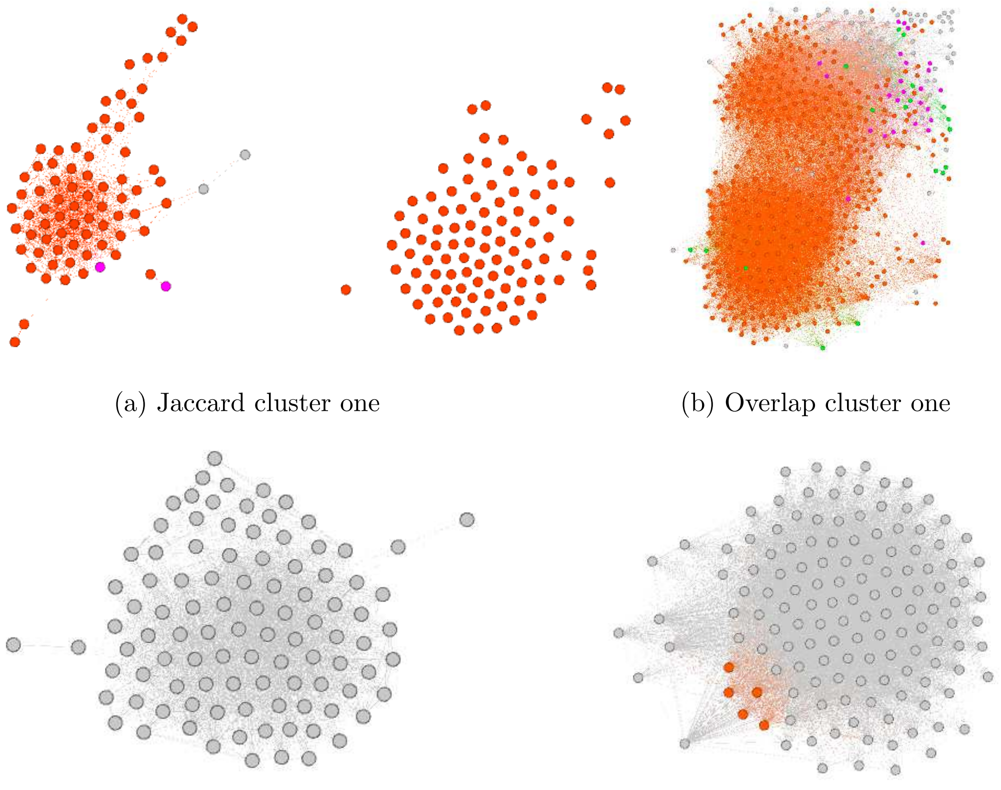

(a) Jaccard cluster one
(b) Overlap cluster one
(c) Jaccard cluster two
(d) Overlap cluster two

Figure 6.15: Side by side comparison of clusters for dataset three

The other cluster selected for evaluation is the circular one found top-right in the figure, which contains almost no known smart contract interfaces, apart from 4 "Token" interfaces. The cluster has 137 nodes, 8 090 edges, and its average edge weight 0.68, which is the highest for this dataset.

Comparative analysis of clusters

Figure 6.15 shows a side-by-side comparison of the clusters produced by Jaccard and Overlap similarity respectively.

The first image, or figure 6.15 (a) is comprised of the two different clusters because all of the interfaces contained in those two clusters are also contained in the large one produced by Overlap with the same similarity cut-off.

Similar to the previous two datasets, both clusters produced by running the Overlap algorithm are larger, denser, and have higher average edge weights, or 0.62 and 0.68, whereas Jaccard's clusters all have an average edge weight of 0.61. The difference is not that large when only this parameter is observed. However, that changes when the density of the clusters is taken into consideration. Overlap's clusters have nodes to edges ratio of 430 : 30 206 and 137 : 8 090, while Jaccard's have 99 : 2 679, 100 : 3 141, and 78 : 1 056. When these numbers are summarized, the ratio of clustered data for the graph produced by Overlap is 567 : 38 296 and for the one produced by Jaccard is 277 : 6 876. Furthermore, when comparing the second pair of clusters in the figure, we can notice that the one produced by Overlap is able to find known interfaces, which give us a better idea of the functionality of the cluster. The 4 known smart contract interfaces in these clusters have the label "Token".

On the other hand, Jaccard similarity is able to produce clearer and more concise clusters, containing highly homogeneous data. That means, that these clusters are more likely to contain smart contract interfaces of the same type, or with the same labels and/or sub-labels. Even when the similarity cut-off is reduced to 0.1, the "Token" interfaces tend to move towards the same cluster, while the others move away from it, which is not the case when Overlap similarity is implemented. Finally, when similarity cut-off is reduced to 0.1 on the Overlap algorithm, no clusters can be observed, the same as the previous dataset.

6.4 Discussion

This section covers insights that we deducted from the evaluation of the three distinct datasets in regards to their produced clusters by using two different clustering approaches. When Jaccard or Overlap similarities are implemented on distinct smart contract interfaces in the Ethereum network, clusters of similar interfaces are produced. In each of the selected datasets for evaluation, at least two large clusters can be noticed, with a minimum average edge weight of 0.6. In regards to the parameters chosen for evaluating the quality of the clusters, the following observations were made:

- **Average edge weight of the clusters**

  The largest average edge weight in a cluster was detected in one of those produced by the Overlap similarity algorithm, which stands at 0.8. This cluster can be observed in Figure 6.5 (b).

  On the other hand, the lowest average edge weight noticed throughout the experiments, for a similarity cut-off of 0.5, was 0.61, and was found in three different clusters. All of them are produced by running the Jaccard similarity for the third dataset and can be observed in Figure 6.15 (a) and (c). Clusters with smaller average weights were noticed as the similarity cut-off was lowered.

  The situation is similar to the rest of the clusters. As expected, Overlap similarity produces clusters with a higher average edge weight, as the value in its denominator is smaller than the one for Jaccard.

- **Size and Density of the clusters**

Overlap similarity has an overall better performance in regards to the nodes to edge ratio and usually produces denser clusters. The highest ratio found was in one cluster produced by the Overlap algorithm, and stands at 364 : 54 499. This cluster can be observed in Figure 6.9 (b).

As expected, the cluster with the lowest nodes to edges ratio was one of the three clusters produced by Jaccard in the last evaluated dataset and has a value of 78 : 1056. This cluster can be observed in Figure 6.15 (a), left.

However, the density of the clusters can be changed by using lower similarity cut-offs. This configuration works properly only for graphs produced by the Jaccard similarity algorithm, which even with a low similarity cut-off can create clusters. When this configuration is used on the Overlap algorithm, the whole dataset is clustered together, and no knowledge can be gained. An example of such a graph, created by using Jaccard can be observed in figure 6.13.

- **Ratio of known smart contract interfaces of a certain type to others in a single cluster**

  When it comes to detecting more concise and homogeneous clusters, Jaccard similarity performs significantly better, for all three datasets. This algorithm was able to detect 1 cluster that is 100% homogeneous, which can be observed in figure 6.15 (a), right, and 4 more which are more than 90% homogeneous.

  One of the clusters from the first dataset produced by Overlap performed the worst in regards to this evaluation metric and can be observed in figure 6.5 (d). In this cluster, the Overlap algorithm finds similarities between two different types of known smart contract interfaces. It contains 19 "Wallet" and 18 "ercToken" interfaces. Wallet and token smart contracts are not "completely" different, as the purpose of many wallets is to handle tokens. However, the overlap may be considerably small (just 1 to a few signatures) and also depends on the size of the wallet interface.

- **Ratio of known smart contract interfaces of a certain type inside to those outside a single cluster**

  In regards to this last evaluation metric, two observations can be made. First, when the similarity cut-off is set to 0.5, Overlap performs better. An obvious example is the clusters in the third dataset, for which Jaccards separates "Token" interfaces into two large clusters while Overlap combines them into one. This difference in clusters can be seen in figure 6.15 (a) and (b).

  The second observation is that when the similarity cut-off is reduced to 0.1, Jaccard is able to combine the majority of the same labeled interfaces into a single cluster, while Overlap clusters the entire data into a single cluster. The difference between these two algorithms for a low cut-off can be observed in figure 6.10. However, due to size and density, these graphs do not produce clear images.

CHAPTER 7

Conclusion and Future Work

In this final chapter and the next several sections, a summary and a conclusion are provided, starting with the contribution of the thesis, a discussion regarding the research questions presented in section 1.1, and finally concluding with limitations and challenges, and possibilities for future work.

7.1 Summary and Contribution

This thesis represents one step towards the bigger picture understanding smart contracts and how they work together in a blockchain network. Following this, two contributions from this research can be recognized:

- **Identify and understand smart contracts that share functionality**

  The first and most important contribution of this thesis is the research to understand how Ethereum smart contracts that have the same or similar functionality are connected in the network.

  By creating clusters of Ethereum smart contracts using two different similarity algorithms implemented by the graph database Neo4j, in regards to the function they share, we were able to identify groups of smart contracts with the same or similar functionality. These clusters were later loaded to a large-scale network visualization tool, Gephi, for a better overview and comparison of the clusters yielded from each of the similarity algorithms.

- **Help identify and label smart contracts**

  Another contribution of this thesis is assisting in the ongoing research for identifying and/or labeling unknown smart contracts, thus continuing the work of numerous scientists and researchers in the area of understanding Ethereum smart contracts.

Examples of such research include Di Angelo *et al.* [dAS19] and Norvill *et al.* [NFS+17], that have already been covered in chapter 3.

To that end, labels of known smart contracts were loaded into the preexisting dataset. There are several benefits to this approach. First, by loading labels of known smart contracts to the already created clusters, we were able to better evaluate the validity of our results. If a certain label (or type of label) is completely contained in one cluster, we can be sure that the clustering algorithm has produced valid results. Furthermore, it was also helpful in the comparative analysis of the two different Neo4j similarity algorithms, Jaccard and Overlap. Finally, if a group of labeled smart contracts is found in a cluster, it can be later used to identify new ones.

7.2 Discussion of Research Questions

The two research questions, formulated in section 1.1, help to develop an understanding of smart contracts in the Ethereum environment, and at the same time guided the course of this thesis. This section provides answers to those questions.

- **RQ1: Without prior knowledge about the contracts, which structures/ patterns can be observed in the contract graph?**

  Structures in the form of clusters can be detected in almost every connected component in the dataset, with their size and density depending size of the dataset and the selected clustering approach. Numerous distinct smart contract interfaces share functionality and depending on the similarity algorithm chosen, and its configurations, different types of clusters are created, as it can be observed in tables 6.2 and table 6.4. Generally, clusters produced by the Jaccard similarity algorithm are clear and concise, with a clear separation from the rest of the dataset loaded in Gephi, which is especially noticeable when a higher similarity cut-off is configured for the algorithm. However, sometimes the structures are very small (from 5 to 10 nodes) and they are not considered as clusters in the information tables. When this cut-off is reduced, the clusters increase in size, however, the line of separation between pieces of clustered data becomes blurrier. When the size of the evaluated dataset is larger Jaccard similarity with higher cut-off produces better results and vice versa. On the other hand, the graphs produced by the Overlap similarity algorithm, even with higher similarity cut-off detect larger and denser clusters, in which the clustered and unclustered data is not clearly separated. When the similarity cut-off is reduced, it is not uncommon for the whole dataset to cluster together, and no structures can be noticed. Since Overlap is able to find a lot more similarities between the data in a single connected component (enough to cluster the entire datasets in some cases), it leads us to conclude that a lot of smart contracts in the Ethereum ecosystem are subsets of others, in regards to the ABI function signatures they share.

- **RQ2: With prior knowledge about the contracts, which structures/ patterns can be observed in the contract graph?**

  Once additional knowledge has been added to the entire Ethereum dataset in the form of known/labeled smart contracts, a large portion of the structured data gains more meaning. We can see if the known smart contract interfaces are pulled together in clusters, how many of them gravitate towards a single cluster in the dataset, what portion of unknown interfaces are closely clustered with them, if different types of known interfaces can be found is a single cluster, and gain more insight in the quality of clusters in regards to the chosen clustering approach. Both clustering approaches produce reliable results for the three selected datasets, and it can be concluded that usually known smart contract interfaces of the same type are clustered together. However, when it comes to concise clusters with clear lines of separation, Jaccard similarity produces better results, and this superiority is proven by the high ratio of known interfaces to others inside a single cluster. On the other hand, Overlap similarity produces denser clusters that attract a larger number of nodes, which leads us to believe that a significant number of nodes are subsets of others in regards to their functionality (as mentioned before, Overlap similarity is usually used to work out which things are subsets of others).

7.3 Limitations and Future Work

This section covers first, the limitaions and challenges of this thesis that in one way or another stopped us form getting more refined results, and second, the possibilites of future work and research that could improve on what we have built so far.

The limitations encounterd during the course of this research are the following.

- **Memory and resources**

  The challenges with resource allocation have already been mentioned during the course of this thesis. In section 4.1 and section 4.2 several possible solution were presented and explained. Some of those approaches were implemented, like using a larger server (provided by TU Wien), exporting some of the logic to a Spring Boot project, and partitions the data.

  However, even with the steps taken to reduce the resource requirements, this limitation has still proved to be the most significant one. The size of the data is simply too large for a single machine with average settings and configurations to handle.

- **Lack of labeled smart contracts**

  One important approach in this research for gaining a deeper understanding of smart contracts in the Etherem ecosystem was the comparison of clustered contracts to already known and/or labeled ones. As already mentioned, this approach was key for the comparative analysis of the different similarity algorithms that were implemented.

However, in the current phase of blockchain and smart contract research, the number and quality of labeled and known smart contracts are limited and not reliable enough to make a big difference in the process of evaluating and comparing results.

These limitations, noted above, offer opportunities for future research on the topic. First of all, using some of the more expensive measures and approaches to tackle challenges of resource allocation, explained in section 4.1, far better clusters of smart contracts that share functionality can be produced. This applies both for the clustering algorithms themselves, which on better and/or more machines can handle and manipulate large sets of data (also faster), and for visualizing and manipulating clusters in network visualization tools. The latter is especially significant because there is just so much data Gephi can handle with average settings on a normal machine.

Finally, new research and studies for the Ethereum blockchain and Ethereum smart contracts keep appearing, and considering the popularity of the topic, they will not stop any time soon. Some of them, similar to this thesis, help in the work of understanding, identifying, and labeling unknown smart contracts. Once a larger portion of all smart contracts in the Ethereum network are identified and deemed as "known", this clustering model will produce far better and more concise results. Once more known smart contracts exist, different clustering approaches and algorithms can be used and compared, and with that, an optimal approach can be found.

Acronyms

ABI	Application Binary Interface. 6, 17, 27, 42, 59, 82
BI	Business Intelligence. 29, 30
CCG	Contract Creation Graph. 21
CIG	Contract Invocation Graph. 21
CPU	Central Processing Unit. 29
CRUD	Create, Read, Update, and Delete. 34
CSV	Comma-separated values. 33, 34, 38, 42, 45, 49, 51, 52
dApp	Decentralized application. 2, 17
DSL	Domain Specific Language. 46
EOA	Externaly Owned Accounts. 16, 21
EVM	Ethereum Virtual Machine. 15–17
FinTech	Financial Technology. 17
GUI	Graphical User Interface. 30, 41
ICO	Initial Coin Offering. 18, 20
IDE	Integrated Development Environment. 37
IoT	Internet of Things. 17
JVM	Java Virtual Machine. 36, 45
MFG	Money Flow Graph. 21

OGM	Object-Graph Mapping. 46, 89
PoA	Proof of Authority. 11
POJO	Plain old Java object. 45
PoS	Proof of Stake. 11
PoW	Proof of Work. 10
SCC	strongly connected components. 21
SLF4J	Simple Logging Facade for Java. 37
SQL	Structured Query Language. 34
WCC	weakly connected components. 21, 35, 49–51, 58, 59, 62, 89

List of Figures

List of Tables

Bibliography

[abi19] Contract abi specification. `https://solidity.readthedocs.io/en/latest/abi-spec.html`, 2019.

[AH19] Manar Abdelhamid and Ghada Hassan. Blockchain and smart contracts. In *Proceedings of the 2019 8th International Conference on Software and Information Engineering*, ICSIE '19, pages 91–95, New York, NY, USA, 2019. ACM.

[apa14] `http://spark.apache.org/`, 2014.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[BH16] Erwin Filtz Bernhard Haslhofer, Roman Kar. O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs. `https://pdfs.semanticscholar.org/96b9/5da0ab88de23641014abff2a5c0b5fec00c9.pdf`, 2016.

[BHJ09] Mathieu Bastian, Sébastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 01 2009.

[BKB$^+$07] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571 – 583, 2007. Software Performance.

[BLMR14] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract]y. *SIGMETRICS Perform. Eval. Rev.*, 42(3):34–37, December 2014.

[But15a] Vitalik Buterin. A next generation smart contract and decentralized application platform. `http://blockchainlab.com/pdf/Ethereum_` white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf, 2015.

[But15b] Vitalik Buterin. On Public and Private Blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/, 08 2015.

[But15c] Vitalik Buterin. On Public and Private Blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/?fbclid=IwAR1sUIx9A7ad98tRJcWfJ9RvWEhWvUpeLAQr9gUss1IH2YO_hygNDunJvxc, 08 2015.

[clo] https://console.cloud.google.com/marketplace/details/ethereum/crypto-ethereum-blockchain.

[CO17] W. Chan and A. Olmsted. Ethereum transaction graph analysis. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 498–500, Dec 2017.

[coi17] $7 Million Lost in CoinDash ICO Hack. https://www.coindesk.com/7-million-ico-hack-results-coindash-refund-offer, 07 2017.

[com] Ethereum community. Introduction to Smart Contracts. https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html.

[com17] Ethereum community. Ethereum Homestead Documentation. https://www.readthedocs.org/projects/ethereum-homestead/downloads/pdf/latest/, 2017.

[CZL$^+$18] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange. Understanding ethereum via graph analysis. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1484–1492, April 2018.

[dAS19] Monika di Angelo and Gernot Salzer. Mayflies, breeders, and busy bees in ethereum: Smart contracts over time. In *Proceedings of the Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts*, BCC '19, pages 1–10, New York, NY, USA, 2019. ACM.

[Dav20] Aran Davies. 5 Best Smart Contract Platforms for 2020. https://www.devteam.space/blog/5-best-smart-contract-platforms, 2020.

[DFZ16] Tuyet Duong, Lei Fan, and Hong-Sheng Zhou. 2-hop blockchain : Combining proof-of-work and proof-of-stake securely. 2016.

[DSUBGV+10] David Dominguez-Sal, P. Urbón-Bayes, Aleix Giménez-Vañó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, Josep-Lluis Larriba-Pey, Heng Shen, Jian Pei, M. Özsu, Lei Zou, Jiaheng Lu, Tok-Wang Ling, Ge Yu, Yi Zhuang, and Jie Shao. Survey of graph database performance on the hpc scalable graph analysis benchmark. pages 37–48, 07 2010.

[FA18] Md Abdul Motaleb Faysal and Shaikh Arifuzzaman. A comparative analysis of large-scale network visualization tools. pages 4837–4843, 12 2018.

[FD07] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315 5814:972–6, 2007.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[FV15] Vitalik Buterin Fabian Vogelsteller. ERC20 token standard. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`, 2015.

[gat16] Gatecoin Claims $2 Million in Bitcoins and Ethers Lost in Security Breach. `https://www.coindesk.com/gatecoin-2-million-bitcoin-ether-security-breach`, 05 2016.

[HD03] Mark Huisman and Marijtje Duijn. Stocnet: Software for the statistical analysis of social networks. *Connections*, 25:7–26, 01 2003.

[Her19] Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019.

[Jac01] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:241–72, 01 1901.

[JCw+18] Shan Jiang, Jiannong Cao, Hanqing wu, Yanni Yang, Mingyu Ma, and Jianfei He. Blochie: a blockchain-based platform for healthcare information exchange. 04 2018.

[JV13] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*, pages 708–715, 2013.

[JVHB14] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9:e98679, 06 2014.

[KMN+02] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, July 2002.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[LA12] Antoine Lambert and David Auber. Graph analysis and visualization with tulip-python. 08 2012.

[Lem17] Victoria Lemieux. Blockchain and distributed ledgers as trusted record-keeping systems: An archival theoretic evaluation framework. 11 2017.

[Loo16] Peter Loop. Blockchain: The Next Evolution of Supply Chains. `https://www.mhlnews.com/global-supply-chain/article/22052455/blockchain-the-next-evolution-of-supply-chains`, 2016.

[MBKB11] Shawn Martin, W. Brown, Richard Klavans, and Kevin Boyack. Openord: An open-source toolbox for large graph layout. *Proc SPIE*, 7868:786806, 01 2011.

[Mer80] Ralph Merkle. Protocols for public key cryptosystems. pages 122–134, 04 1980.

[MF14] Sudeep Pillai Michael Fleder, Michael S. Kester. Bitcoin Transaction Graph Analysis. `https://people.csail.mit.edu/spillai/data/papers/bitcoin-transaction-graph-analysis.pdf`, 2014.

[ML12] Julian McAuley and Jure Leskovec. Discovering social circles in ego networks. 8, 10 2012.

[MSuR15] Fariha Majeed and Dr Saif-ur Rahman. Graph visualization tools: A comparative analysis. *Journal of Independent Studies and Research - Computing*, 13, 01 2015.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008.

[neo15a] `https://boltprotocol.org/`, 2015.

[neo15b] https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/, 2015.

[neo16a] http://neo4j.com/, 2016.

[neo16b] https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/similarity/, 2016.

[neo16c] https://neo4j.com/developer/apache-spark/, 2016.

[neo16d] https://neo4j.com/docs/graph-algorithms/current/algorithms/community/, 2016.

[neo16e] https://neo4j.com/docs/, 2016.

[neo16f] https://neo4j.com/developer/guide-import-csv/, 2016.

[neo20] https://techcrunch.com/2020/02/04/neo4j-4-0-graph-database-platform-brings-unlimited-scaling/, 2020.

[NFS+17] R. Norvill, B. B. Fiz Pontiveros, R. State, I. Awan, and A. Cullen. Automated labeling of unknown contracts in ethereum. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, July 2017.

[NH17] Moritz Petersen Niels Hackius. Blockchain in Logistics and Supply Chain: Trick or Treat? https://pdfs.semanticscholar.org/7752/f1275da69d208e5a76d7adc6b12b3b61699e.pdf, 2017.

[NK18] Giang-Truong Nguyen and Kyungbaek Kim. A survey about consensus algorithms used in blockchain. *Journal of Information processing systems*, 14(1), 2018.

[ope] https://github.com/gephi/gephi/wiki/OpenOrd.

[OS10] Chitu Okoli and Kira Schabram. A guide to conducting a systematic literature review of information systems research. *SSRN Electronic Journal*, 10, 05 2010.

[Pil15] Marc Pilkington. Blockchain technology: Principles and applications. 2015.

[PPEKI] Georgios A. Pavlopoulos, David Paez-Espino, Nikos C. Kyrpides, and Ioannis Iliopoulos. Empirical comparison of visualization tools for larger-scale network analysis. *Advances in Bioinformatics*.

[RCC$^+$04] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101:2658–63, 04 2004.

[Ros17] Ameer Rosic. What is An Ethereum Token: The Ultimate Beginner's Guide. `https://blockgeeks.com/guides/ethereum-token`, 2017.

[She19] Benjamin Sherry. What Is an ICO? `http://investopedia.com/news/what-ico`, 2019.

[sim07] Measuring semantic similarity between words using web search engines. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 757–766, New York, NY, USA, 2007. Association for Computing Machinery.

[spra] `https://spring.io/projects/spring-data-neo4j`.

[sprb] `https://docs.spring.io/spring-data/neo4j/docs/5.2.6.RELEASE/reference/html/#repositories.core-concepts`.

[spr13] `https://start.spring.io/`, 2013.

[Sun18] Flora Sun. A Survey of Consensus Algorithms in Crypto. `https://medium.com/@sunflora98/a-survey-of-consensus-algorithms-in-crypto-e2e954dc9218`, 04 2018.

[Sza96] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18:2, 1996.

[VL19] Friedhelm Victor and Bianca Katharina Lüders. Measuring ethereum-based erc20 token networks. In *International Conference on Financial Cryptography and Data Security*, 2019.

[whi16] Whitepaper:Nxt. `https://nxtwiki.org/wiki/Whitepaper:Nxt`, 2016.

[XWS$^+$17] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. 04 2017.