TU WIEN Informatics

Erkennung verwandter Smart Contracts anhand des Bytecodes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Tan Yücel, BSc
Matrikelnummer 01525681

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Gernot Salzer
Mitwirkung: Ass.Prof. Dr. Monika di Angelo

Wien, 20. April 2022

Tan Yücel   Gernot Salzer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Identifying Related Smart Contracts by their Bytecode

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Tan Yücel, BSc
Registration Number 01525681

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Gernot Salzer
Assistance: Ass.Prof. Dr. Monika di Angelo

Vienna, 20th April, 2022

Tan Yücel	Gernot Salzer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Tan Yücel, BSc

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, 20. April 2022

Tan Yücel

Acknowledgements

"I haven't actually seen further, but standing on the giants' shoulders was fun anyways."

— Tan Yücel, *2022*

Kurzfassung

Das wachsende Interesse an Smart Contracts (Blockchain Programmen), die stetig steigende Zahl von Teilnehmern im Crypto-Ökosystem, sowie neuartige Geschäftsideen tragen dazu bei, dass das Gesamtsystem immer komplexer wird. Da nur wenige Entwickler den Quellcode ihrer Smart Contracts offenlegen, kann die Identifizierung von funktionalen Ähnlichkeiten zwischen Smart Contracts nützlich sein, insbesondere der Vergleich von quelloffenen Contracts mit solchen, die es nicht sind. Dabei stellt die automatisierte Analyse von Programmen, die nur als Maschinenprogramm vorliegen, ein immer noch aktives Forschungsgebiet dar.

Die vorliegende Arbeit untersucht ein Verfahren zum Erkennen von Ähnlichkeiten zwischen Smart Contracts, die als Maschinenprogramm vorliegen. Sie baut auf einer Publikation von Huang et al. [HHY+21] auf, die Methoden des Machine Learning, Taint-Analyse und simulierte Bytecode-Ausführung kombiniert. Ausgehend von Codesegmenten, die eine Sicherheitslücke enthalten, extrahieren die Autoren sogenannte *Slices* und repräsentieren sie als numerische Vektoren. Durch den Vergleich dieser mit analog kodierten Smart Contracts gelingt es, ähnliche Sicherheitslücken in anderen Programmen zu identifizieren.

Unser primäres Ziel ist es, die Arbeit von Huang et al. nachzuvollziehen und mit eigenen Datensätzen zu überprüfen. Eine Schwierigkeit besteht dabei darin, dass Huang et al. ihr Verfahren nur lückenhaft beschreiben und ihre Daten nicht öffentlich verfügbar sind. Unsere Ergebnisse sind daher nicht direkt vergleichbar, unsere Experimente liefern aber Anhaltspunkte dafür, dass unsere Rekonstruktion weitgehend dem ursprünglichen Verfahren entspricht. Weiters schlagen wir eine Reihe von Verbesserungen vor.

Ein weiteres Ziel unserer Arbeit ist die Erweiterung des Verfahrens von Huang et al., um die Ähnlichkeiten zwischen Smart Contracts als Ganzes zu bestimmen. Die zu diesem Zweck entwickelte heuristische Matching-Methode vergleichen wir mit etablierten Metriken wie dem *Jaccard Index der Funktionssignaturen* der Contracts. Versuche, die mittels Datensätzen bestehend aus Wallet-Contracts durchgeführt wurden, zeigen auf, dass eine mittelgroße Korrelation zwischen diesen Ähnlichkeitsmaßen besteht.

Abstract

Smart contracts (blockchain programs) have now attracted significant interest, and the growing number of participants in the ecosystem as well as refined business cases add layers of complexities to the system. Automated tools quickly reach their limits when trying to make sense of closed source smart contracts. As only a small proportion of live smart contracts provide their source openly, it can be helpful to automatically determine the functional similarity between smart contracts, especially with open and closed source.

In this thesis, we attempt to identify related smart contracts by extending the work of [HHY$^+$21] who tried to detect vulnerabilities in smart contracts using a combination of machine learning techniques, taint analysis and simulated bytecode execution. By extracting segments of bytecodes into so-called *slices* and calculating their numerical vector representations, the authors were able to compare parts of smart contracts with each other. Applying this on a set of vulnerable contracts allowed for the detection of priorly unknown vulnerabilities in other smart contracts.

The primary goal of this work is to create suitable datasets and verify the methods deployed by [HHY$^+$21]. While our results do not reach the levels of [HHY$^+$21], we believe that they show that our implementation of the method works as intended. Based on our findings, we compile a list of suggestions for substantial improvements in future iterations.

The second goal of this thesis is to extend the above-mentioned method so that it can also detect similarities between contracts as a whole. We deploy a scalable heuristic method of matching several slices of different contracts with each other. By comparing our computed similarity score with existing metrics, e.g. the *Jaccard index over function signatures* between two contracts, we try to argue in favor of our extension of the original method. Experiments conducted over a dataset consisting of wallet contracts show that a moderately high correlation between our method and the Jaccard index can be achieved.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The rise in popularity of blockchain technologies in the last decade has spawned many interesting research areas. Smart contracts (blockchain programs) in particular have attracted significant interest, as advocates claim revolutionary innovations for the financial sector by eliminating the need for trust from business processes. The ability of smart contracts to interact with each other, as well as the growing number of participants in the ecosystem, pave the way for increasingly refined business cases which add layers of complexities to the system that become increasingly difficult to unravel. Existing automated tools often reach their limits when trying to make sense of closed source smart contracts.

The nature of smart contracts on the Ethereum blockchain makes understanding their semantics quite a difficult task when the source code is not available. While some developers cooperate by allowing external audits, or by disclosing the source code of their dApp (and thereby putting their business model at risk), most smart contract creators do not act in such good faith. As previous work has shown, source codes are only directly available for 11 % of the smart contracts [dAS19b]. But even with source code available, it remains a non-trivial task to detect functional similarities automatically on a large scale.

Another dimension is added to this problem by the most widespread compiler, Solidity Compiler (solc)[1], which offers different levels of gas cost optimization. This fact, combined with the continuous evolution of the compiler over the last 7 years, makes it difficult to relate smart contracts to each other without relying on information from outside the blockchain (e.g. the contract source code).

[1]https://docs.soliditylang.org/en/v0.4.21/installing-solidity.html

We hope that this work will help paint a clearer picture of the smart contract landscape on Ethereum blockchain. Our findings may prove relevant to people who want to know more about *how*, *where* and *which* contract code is re-used, and at which scale it is happening on the Ethereum blockchain.

1.2 Problem Statement

Huang et al. [HHY^{+}21] describe a promising method for detecting vulnerabilities in smart contracts by combining machine learning algorithms, taint analysis and other techniques to extract information out of contract bytecodes. By doing so, they claim that they were able to automatically detect vulnerabilities that were previously undiscovered.

The aim of our work is two-fold. On the one hand, we want to reproduce the findings of [HHY^{+}21], in order to confirm the suitability of the proposed methods for vulnerability detection. The difficulties herein lie in the fact that the authors have neither published any source code, nor any of the datasets they operate on. This means, a software prototype needs to be implemented, and suitable datasets created according to their description. In addition, many assumptions have to be made at places where the original paper lacks in detail. By establishing a working software pipeline and selecting adequate datasets, we verify their method.

On the other hand, we want to investigate whether the methods used by [HHY^{+}21] can be leveraged to relate entire contracts to one another. Our reasoning is that if their work can be used to identify vulnerable bytecode within bodies of contracts, it should be adequate for the detection of similarities between contracts as a whole. In summary, our second goal is to measure how well the methods that we modelled after [HHY^{+}21] are suited to detect semantic similarities between smart contracts en masse.

1.3 Research Questions

The term *related contract* in the title leaves room for interpretation. One possible understanding is the sense of contracts belonging together, e.g. smart contracts that can only act out a certain process when they are combined, and use each other to process a transaction. *Related* in the more traditional sense could also imply contracts that are literal “offsprings” of other contracts, as is observable with *breeder* contracts [dAS19b]. The thesis will focus on a third interpretation: Related contracts in the sense of contracts sharing (parts of) the same semantics. More precisely, we will focus on related contracts in the context of the following research questions.

RQ$_1$: Can we verify that the method of [HHY^{+}21] performs as described when used on our dataset?

By creating a prototypical implementation of their described method and using comparable datasets, we aim to verify whether their methods are expedient or not.

RQ$_2$: How well can semantic similarities between smart contracts be detected by extending the method of [HHY$^+$21]?

By extending their method of comparing individual slices, we want to relate entire contracts to each other. For this we make use of known to be similar contracts like the Ethereum Request for Comment (ERC) token suite, or wallet contracts that were previously classified by [dAS20d]. We regard the function signatures of smart contracts as points of orientation when determining the quality of our results.

1.4 Related Work

To a large extent, our work is based on the findings of [dAS19b], who investigated the similarity of contracts based on their function interfaces.

EtherSolve — a tool developed by Contro et al. [CCCP21] — computes the Control Flow Graph (CFG) of a smart contract starting from its creation or runtime code. The authors released their findings as part of an open-source GitHub repository[2]. EtherSolve includes a command line interface and is well documented. We will use it for various analysis tasks, as well as a building block to obtain CFGs.

Narayanan et al. [NCV$^+$17] introduce graph2vec, a neural embedding framework that is able to create vector representations of arbitrarily sized graphs in an unsupervised manner. The authors of [HHY$^+$21] leverage this framework to efficiently detect similarities between CFGs of smart contracts. By doing so, they aim to detect commonly occurring vulnerabilities in smart contracts — an approach that is also carried out by us.

In a related note, Xu et al. [XLF$^+$17] introduced a novel approach to detect similarities in *conventional* binary code using graph embeddings. They published their software prototype on an open source repository[3], though a plugin for the binary code analysis tool IDA[4], which is used to obtain the graph embeddings from the program binaries, is not publicly available.

Osiris is a framework by Ferreira Torres et al. [FTSS18] based on a procedure similar to [HHY$^+$21] combining symbolic execution and *taint analysis*. In their work, the authors have been successful in finding integer bugs in smart contracts.

Fröwis et al. [FFB19] use similar approaches of symbolic execution and taint analysis to accurately classify Token contracts as such. Their approach is able to "*[...] detect token systems by their characteristic behavior of updating internal accounts*" [FFB19, p. 93].

Liu et al. [LYJ$^+$19] introduce the notion of smart contract birthmarks, a "*semantic-preserving and computable representation for smart contract bytecode*" [LYJ$^+$19, p. 105]. These birthmarks consist of a combination of syntactic features, as well as some semantic patterns defined by the order of appearance of opcodes within single building blocks in

[2]https://github.com/SeUniVr/EtherSolve
[3]https://github.com/xiaojunxu/dnn-binary-code-similarity
[4]https://hex-rays.com/ida-free/

a smart contract's CFG. Using these birthmarks, the authors of [LYJ+19] were able to implement a clone detector achieving high levels of accuracy on an undisclosed dataset. The authors provide their software prototype in the form of a public GitHub repository[5].

Eth2Vec, a machine-learning-based static analysis tool by Ashizawa et al. [AYCO21], is able to identify smart contract vulnerabilities by its bytecode. The authors claim that *Eth2Vec* is even robust against rewrites, thus detecting vulnerabilities more reliably. *Eth2Vec* uses a neural network for natural language processing, which allows to learn the context of each function in order to extract features more precisely. Its source code is available on a public GitHub repository[6].

Mythril, a security analysis command line tool developed and maintained by the company *ConsenSys*, is able to execute smart contract bytecode symbolically and visualize the CFG of smart contracts [Ber18]. By making use of techniques such as SMT solving and taint analysis, *Mythril* is able to detect a variety of security vulnerabilities in smart contracts, e.g. reentrancy bugs or unsafe math operations [Con21, dAS19a].

[5]https://github.com/njaliu/ethereum-clone/
[6]https://github.com/fseclab-osaka/eth2vec

CHAPTER 2

Background

A short introduction to blockchain- and other relevant technologies and techniques used in this work is provided in this chapter. Due to longevity reasons, we forgo a detailed workup, especially in regard to technical implementations of core blockchain technologies that are not in the scope of this work. We also omit some details about Neural Networks (NNs), and instead refer to our cited sources, as they represent much more coherently assembled works in this area. Rather, we describe the functional aspects of embedding networks with special focus on *graph2vec* as it was used in the conducted experiments. Should open questions remain beyond what is explained, please refer to the cited sources to gain a more thorough image of the topic.

After covering key aspects of blockchain technologies, this chapter will present peculiarities of smart contracts on Ethereum, the Ethereum Virtual Machine (EVM) as their execution environment, the Solidity programming language, the Solidity compiler *solc*, Control Flow Graphs (CFGs), and finally graph2vec.

2.1 Blockchain Technologies

Hewa et al. [HYL21] name three key features associated with blockchain technologies. Through **decentralization** of compute nodes blockchains achieve high redundancy which essentially guarantees availability. Contributors to the blockchain network share the authority by built-in (governance) protocols. **Immutability** allows a deterministic single source of truth: Participants of the network can verify the integrity of the blockchain data themselves at any time. This is enabled by **cryptographic links** (i.e. hash pointers) that ultimately make for an absolute chronological order of transaction records. Furthermore, participants of the network can verify each other's integrity using digital signature schemes and do not have to *trust* a third party to act as a trustworthy middleman [HYL21]. Transaction validators (i.e. miners) are generally encouraged to participate in such blockchains through monetary *rewards* [Nak08].

Bitcoin, the first public blockchain application, combines these concepts to create a *Peer-to-Peer Electronic Cash System*, as named by the original author [Nak08]. Initially seen as a means to send financial transactions securely around the globe, the paradigm of blockchain has undergone numerous small but significant shifts since then. Many blockchain systems have sprung into existence ever since, bringing, e.g., improved consensus protocols, better scalability, or extensive on-chain self-governance. As already mentioned in chapter 1, our work focuses on the Ethereum blockchain, specifically on its smart contracts.

2.2 Ethereum

Buterin et al. [But14] describe the purpose of Ethereum as follows:

"*The intent of Ethereum is to merge together and improve upon the concepts of scripting, altcoins and on-chain meta-protocols, and allow developers to create arbitrary consensus-based applications that have the scalability, standardization, feature-completeness, ease of development and interoperability offered by these different paradigms all at the same time.*" [But14, p. 12]

While the concept of smart contracts had already been described as early as the 1990s [RBC$^+$98], Ethereum was the first blockchain to design and implement a system that was not only able to execute such contracts, but also to have the characteristics mentioned in section 2.1 [But14].

The Ethereum blockchain stores a *world state* comprising different properties per account. Examples for such properties are e.g. an account's balance or a code section for storing smart contract bytecode. These values can only be manipulated through *transactions*.

2.3 Smart Contracts

Smart contracts (also referred to as *contracts* henceforth) are computer programs that can be stored on the blockchain. Users can interact with them by issuing transactions specifying a contract's address, data determining which function should be executed, as well as the arguments it takes (if any) [But14]. Typical examples for applications of smart contracts are, e.g. Token contracts (Non-fungible Tokens (NFTs)), Decentralized Finance (DeFi) apps, or wallet contracts. Even though smart contracts can interact with each other (for example by calling each other's functions), they do not do so on their own. This makes them *event-driven* — they cannot execute without an external account initiating the call.

Transactions encapsulate the necessary information to compute the next state from the current one. This means that one can compute the most recent world state (comprising all balances, contract storages, etc.) by successively executing all transactions in their respective order in Ethereum's own execution environment, the Ethereum Virtual Machine (EVM) [But14].

In order to mitigate Denial of Service (DoS) attacks, each individual instruction that is executed inside the EVM has a particular cost assigned to it. These costs are referred to as *gas* costs. For a transaction to succeed, a sufficiently large amount of ether has to be provided by the issuer of the transaction. The gas price is paid in ether and its cost generally fluctuates based on the network's load.

2.3.1 Solidity and solc

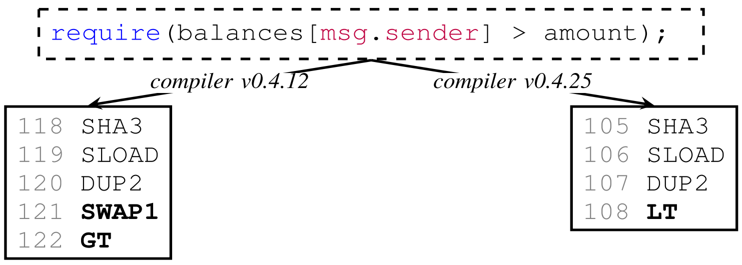
While there are several languages that can be compiled to EVM bytecode such as Vyper or Rust, the dominant platform to write smart contracts on Ethereum is the Solidity language [QTN21]. Its syntax is influenced by C++ and JavaScript and it offers typical Object-oriented programming (OOP)-like patterns, e.g. inheritance or interfaces [Eth22b]. Using Listing 2.1 as an example, we want to highlight some of the language's features which will later turn out to be relevant to this work.

```
 1 pragma solidity >=0.7.0 <0.9.0;
 2 
 3 contract Storage {
 4 
 5     uint256 number;
 6 
 7     constructor(uint _number){
 8         number = _number;
 9     }
10 
11     function store(uint256 num) public {
12         number = num;
13     }
14 
15     function retrieve() public view returns (uint256){
16         return number;
17     }
18 }
```

Listing 2.1: An example smart contract with a constructor.

The first line in this contract constrains the versions of solc that can be used to compile this contract — a feature that was introduced with version 0.4.0. Compiling the same source code using different versions of solc will often result in slightly different bytecodes, as is depicted by Figure 2.1. The cost of deployment of a contract can be seen as a linear function of the size of its bytecode. In order to save cost, the developer can tell solc whether to optimize for a cheap deployment or gas-efficient computing during the contract's lifetime. For this, a parameter estimating the total function calls during a contract's lifetime can be provided to solc using the argument `--optimize-runs`. This feature adds further variation to contract bytecodes that embody the same semantics, and poses another hindrance when trying to relate similar contracts to each other.

Per default, solc will compile contracts into *deployment* bytecode. During deployment, it executes the constructor code (e.g. line 7 in Listing 2.1), and then copies the remaining



```
require(balances[msg.sender] > amount);
```

```
118 SHA3
119 SLOAD
120 DUP2
121 SWAP1
122 GT
```

```
105 SHA3
106 SLOAD
107 DUP2
108 LT
```

Figure 2.1: Different compiler versions resulting in different bytecode. The instructions marked **bold** perform the same semantic operation. Figure taken from [HHY+21].

contract-code onto the blockchain, i.e. to an account's *code* section, to persist it. In the case of our example, the provided argument `_number` is persisted in the storage variable `number` during deployment. This work will focus only on *deployed* contract bytecodes, and we will perform our experiment only on such.

Function Signatures

Functions in smart contracts are identified by the hash of their signature — a characteristic that we benefit from later. To be precise, the first four bytes of the hash are regarded as *unique* enough to adequately identify functions within a single contract [Eth22a]. As it turns out, these 4-byte identifiers encode additional information into contracts when they are used in combination with a mapping between function signatures and their respective 4-bytes, as can be found here for example [dAS19b]. The use of such a mapping allows to reason about the semantics of contracts without necessarily knowing their source code. For a complete specification of how function selectors are computed we refer to [Eth22a].

To give an example, contracts conforming to an ERC token standard can be identified as such using these 4 byte signature hashes [dAS20c]. Naturally, developers can not be forced to adhere to standards when writing contracts — a contract could implement the token interface, but the actual bodies of the functions could be empty, or do something else entirely. Knowing this makes it a bit more challenging to make reliable statements about the semantics of individual contracts, but we believe that function signatures provide enough information to be a good metric when dealing with many contracts at once, especially when trying to reason about their similarities like we will do later.

2.4 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a (quasi) turing-complete stack machine [HSR+18], and as part of the consensus protocol is implemented by every node participating in the Ethereum blockchain [Woo14]. It is responsible for the correct execution and validation of transactions that interact with smart contracts.

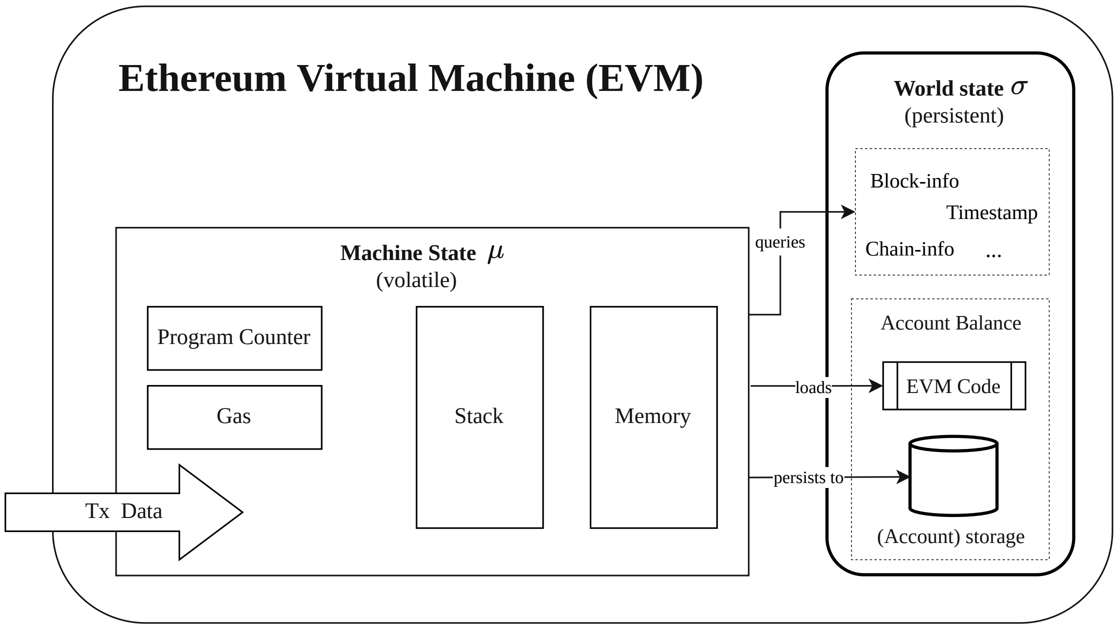


Figure 2.2: Illustration of the EVM and its core components. Figure adapted from the official Ethereum documentation[1].

As visible in Figure 2.2, the EVM's *volatile* machine state $\boldsymbol{\mu}$ consists of the *stack*, a *memory* region, and a Program counter (PC) keeping track of the currently executed instruction. During execution, the EVM has access to the *persistent* world state $\boldsymbol{\sigma}$, which can be used to query information about the execution environment, e.g. using the instructions **BLOCKHASH** or **NUMBER**. Another important section in Figure 2.2 is the arrow depicting the *transaction data*. Using instructions like **CALLER**, **CALLDATALOAD** or **CALLVALUE**, the EVM can query this section for sender provided data. Based on this, the EVM can identify which function to execute, the arguments to use, and the amount of ether provided by the sender. In order to persist data, each smart contract has access to a personal *storage* section. The storage is readable by everyone, but only a contract can write data to its own storage. For each instruction that is executed, the EVM deducts the instruction's individual gas cost from the total gas provided by the transaction sender. As already mentioned, this mechanism protects the network against DoS attacks by preventing transactions to run indefinitely [Woo14].

The EVM's instruction set currently consists of 145 distinct 1-byte wide instructions [Woo14][2] — also referred to as *opcodes*. Besides the above-mentioned instructions to query environmental information, they comprise operations for basic arithmetic and bitwise computation, hashing, logging, and other system functionalities that, e.g., allow interacting with or query information about other smart contracts. In addition to these, further

[1]https://ethereum.org/en/developers/docs/evm/
[2]https://ethervm.io/

instructions exist which may manipulate the world state by creating *new* contracts (**CREATE**, **CREATE2**), or enabling contracts to destroy themselves (**SELFDESTRUCT**) [Woo14]. Transfer of ether (i.e. monetary value) between two accounts (e.g. a contract or externally owned) is realized by using one of the **CALL*** operations or a **CALL** transaction. As the EVM is a *stack machine*, stack manipulating operations of the families **PUSH**, **DUP** and **SWAP** are extensively used [Koo92].

2.4.1 Stack, Storage, Memory

During the execution, the EVM has access to three different kinds of areas where data can be read from and written to.

Stack The stack is ephemeral and interacted with by most of the opcodes. Individual elements in the stack range from one to 32 bytes in size, and the stack can store at most 1024 elements. Depending on the instruction, the number of elements consumed from the stack ranges from 0 to 7, but none of the currently existing instructions produce more than one output element onto the stack [Woo14].

Memory The memory section resembles a computer's memory: Opcodes that use larger amounts of data use this area as a type of buffer. It is a bit more expensive to use than the stack, albeit still transient [Woo14].

Storage The storage is the only persistent area. It is a key-value store of 256-bit length each. Since the storage is replicated along all nodes in the network, taking up space here is an expensive operation in terms of real cost and gas cost. For this reason it is encouraged to free up unused storage by providing a partial refund on the gas that was used when initially storing the data [Woo14].

2.5 Common Types of Contracts

In the context of this work, it is worth mentioning two types of contracts in particular.

Wallet Contracts A wallet contract is described by di Angelo et al. as a "*contract that provides functionality for collecting and withdrawing Ether and tokens via its address*" [dAS20b, p.393]. We will use their findings — wallet contracts classified into groups — as a ground truth to measure the quality of our results in chapter 4.

Standard Contracts Standard contracts are contracts that adhere to a standardized interface. As already mentioned in section 2.3.1 these are identifiable over their function signatures. An example for such contracts would be the ERC contract suite. *SafeMath*, a standard library to prevent integer under- and overflows, was commonly used by other contracts before solc included built-in checks for these types of errors.

2.6 Smart Contract Skeletons

We use the technique of "*skeletizing*" bytecodes described in [dAS19b], [dAS20d] and [dAS20a].

"The skeleton is obtained from the bytecode by removing parts that do not affect functionality: Solidity meta-data, constructor arguments and PUSH arguments are replaced by zeros, then trailing zeros are stripped. [...] Two bytecodes are functionally equivalent if their skeletons are identical." [gsa22]

As we will later describe in more detail, this allows us to reduce the number of *truly* unique contracts when building our datasets. The code used for this is available on GitHub[3].

2.7 Control Flow Graphs

As is the case with conventional compilers, solc compiles smart contracts into modular semantic units with the goal of optimizing for performance, length of the compiled code and, in case of smart contracts, gas cost.

Using symbolic execution, Contro et al. [CCCP21] were able to create CFGs from bytecodes of contracts. Figure 2.3 provides a visual depiction of such a CFG. The authors define basic blocks as "*a sequence of opcodes which are executed consecutively between a jump target and a jump instruction, without any other instruction that alters the flow of control*" [CCCP21, p. 4]. Key to identifying the basic blocks (i.e. nodes) of a CFG are those opcodes that *do* alter the flow of control. The instructions marking the end of a basic block are **JUMP**, **JUMPI**, **STOP**, **REVERT**, **RETURN**, **INVALID** and **SELFDESTRUCT**, whereas the instruction **JUMPDEST** marks its beginning.

Since information about jump targets is not stored in the contract bytecode itself but needs to be computed during the runtime of a contract, the authors of [CCCP21] employ symbolic execution to identify the edges between nodes. They classify different kinds of jumps, which we will omit due to the fact that this distinction is not relevant for our experiments.

2.8 Neural Networks and Embeddings

This section gives a (short) primer to Neural Networks (NNs), vector embeddings, and how the former can be used to obtain the latter. We discuss the advantages and potential usages of these embeddings by using *word2vec* as an example, and proceed to describe *graph2vec*, the embedding framework used in this thesis. Since the area of Machine Learning (ML) has numerous distinctive terms associated with it, we display commonly used terminology in *italics* and refer to Google's ML Glossary[4], should uncertainties remain as to their meaning.

[3]https://github.com/gsalzer/ethutils/tree/main/doc/skeleton
[4]https://developers.google.com/machine-learning/glossary

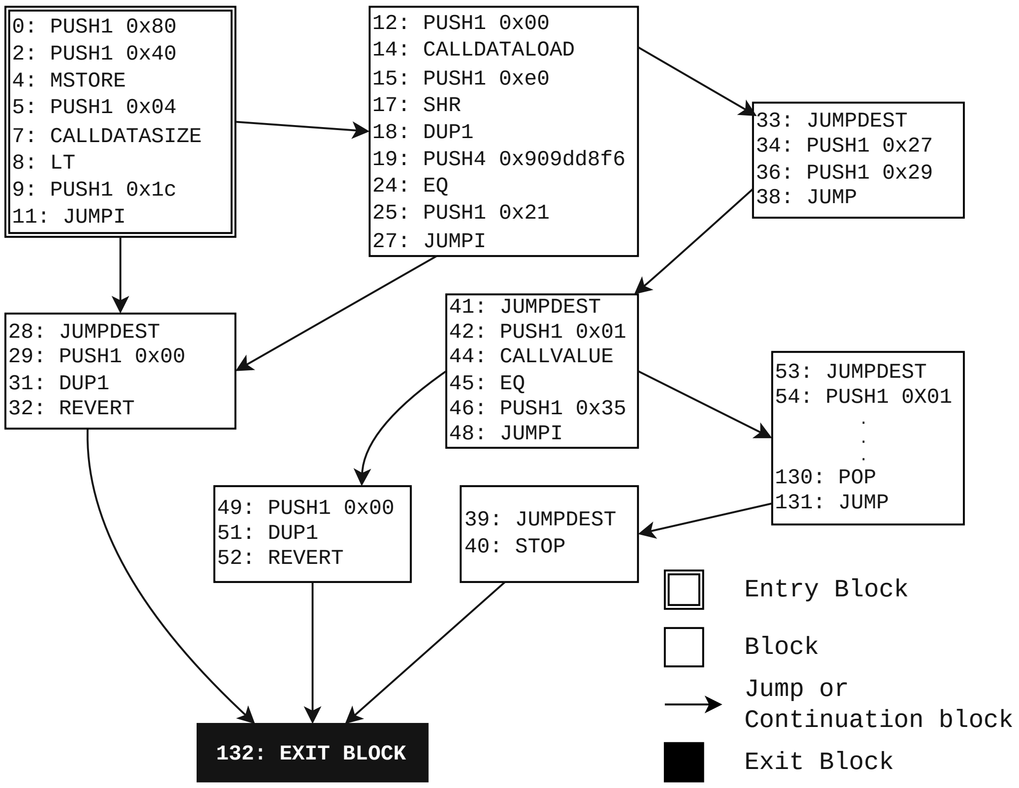
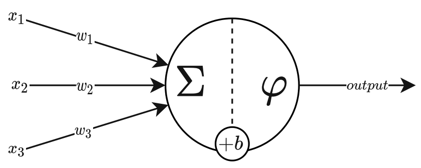


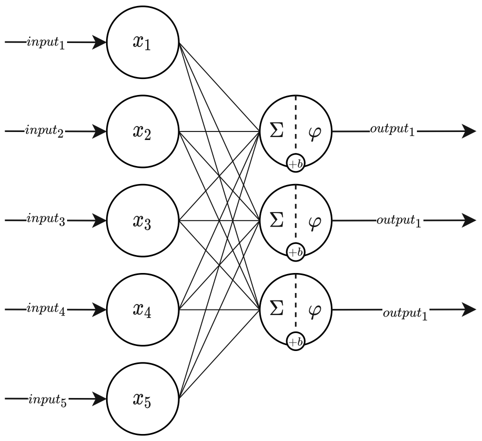
Figure 2.3: Example of an **acyclic** control flow graph. Illustration adapted from [CCCP21].

2.8.1 Neural Networks

The smallest building block of a Neural Network (NN) is called a *neuron*. It is generally described as a data-structure comprising multiple real-valued and individually weighted *inputs* that are summed up and passed to an activation function φ, the neuron's *output* [GK11]. Commonly used activation functions for φ are *Rectified Linear Unit (ReLU)* or the *sigmoid* function [WS21]. Usually, neurons are also assigned a negative or positive *bias* b, which is added to the sum of the weighted inputs. A NN consists of multiple *layers* of such neurons, connecting neurons in layer n with neurons in layer $n + 1$. By varying the number of connections between layers, one can obtain very differently performing models. Examples for common types of layers are e.g. fully connected (*dense*) layers, *convolutional* layers, or *pooling* layers. Since there is no single, general-purpose NN that performs well on every possible task, many different types of NNs have been developed over the past. In the context of embedding tasks, we will focus on *shallow*, *feed-forward* networks, with fully connected layers.



(a) A neuron with 3 inputs and activation function φ. Illustration adapted from [Pic22].



(b) A two-layered, fully connected neural network.

Figure 2.4: Single neuron vs. neural network.

Figure 2.4a provides an example of a neuron with 3 inputs $x_i, i \in \{1, 2, 3\}$, while Figure 2.4b depicts how the outputs of individual neurons are propagated from one layer to another.

Training

Training a NN is the act of adjusting *weights* and *biases* of the network, in a way that improves *inferring* (predicting) *labels* from given *features*. Key to this procedure is a *loss* function, i.e. a method of evaluating how well an algorithm models a dataset. *Backpropagation* is a commonly used technique for *learning* in feed-forward NNs. In the case of embedding networks, the authors of [MCCD13] combine backpropagation with a method called *negative sampling* in order to reduce training time.

2.8.2 Embedding Example: word2vec

The *skip-gram* model introduced by Mikolov et al. [MCCD13] allows to encode words in a N-dimensional vector space while preserving their semantics in an unsupervised manner. It works by training a network using a *one-hot* encoded vector representation of words as feature and their *contexts* as labels. In order to obtain the *word embeddings*, the last layer of the network is dismissed. For each input word, the outputs of the second to last layer are calculated and extracted. These are regarded as the N-dimensional vector representations of the individual words [MCCD13]. Figure 2.5 depicts a network implementing the *skip-gram* model with an embedding dimension of $N = 300$.

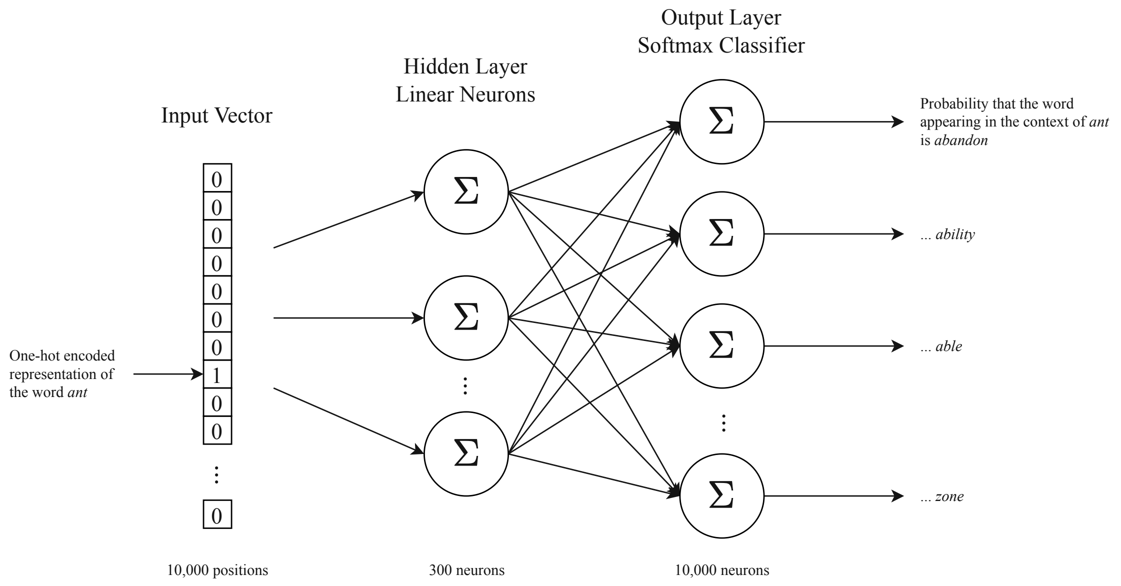
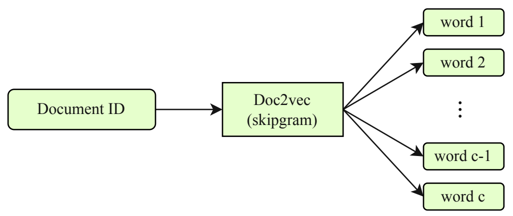


Figure 2.5: Skip-gram as described in [MCCD13]. Image adapted from[5].

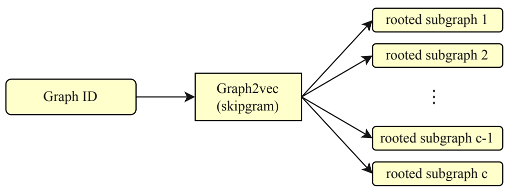
Word embeddings enable interesting applications. A prominent example is one provided by the original authors, and showcases the additive composability of word vectors: "*[...]it was shown for example that vector('King') − vector('Man') + vector('Woman') results in a vector that is closest to the vector representation of the word Queen*" [MCCD13]. In our work we are interested in the peculiarity that the similarity between such vectors in N dimensions is as easy to calculate as it is for vectors in lower dimensions: By computing their cosine similarity. Due to the fact that this is a very light operation, resource-efficient mass computation of similarity becomes viable [MCCD13].

[5]https://israelg99.github.io/2017-03-23-Word2Vec-Explained/

2.8.3 graph2vec



(a) doc2vec samples c words from a document d, and uses them to learn d's representation.



(b) graph2vec treats a graph g as doc2vec treats a document. The rooted subgraphs of g are used as context analogously to words in doc2vec.

Figure 2.6: Comparison doc2vec vs. graph2vec. Both figures adapted from [NCV+17].

The authors of [NCV+17] borrow the idea of unsupervised feature encoding and apply it to graphs. They draw parallels between *doc2vec*, an extension of word2vec for embedding entire documents, and graph2vec. Figure 2.6 illustrates the intuition behind graph2vec. The idea is to treat rooted subgraphs within a graph as words, while the graphs themselves make up the document. According to [NCV+17], this allows for capturing structural equivalences between graphs while learning their embedding vectors in an unsupervised manner. The authors name two use cases for their work, namely graph classification and graph clustering.

The functional aspects of graph2vec that are relevant to our work are that it takes a corpus of graphs as input, and produces their corresponding N-dimensional embedding vectors as output. As was the case with word embeddings, graph embedding vectors also allow for fast comparison by calculating their cosine distance. In this work, we will use these properties to calculate similarities between graphs extracted of smart contracts.

CHAPTER 3

Approach

We base our approach on the work of [HHY+21] by applying the techniques described by the authors. The aim of their work was to automatically detect re-used vulnerabilities among different smart contracts. In addition to trying to recreate their findings, we want to explore if the described methods can be used to detect semantically related bytecode en masse. The difficulty in this task does not only lie in the aforementioned characteristics of the solc compiler, but also in the amount of data that has to be processed. Thus, while often referring to the original, this chapter will also go into detail about the enhancements that had to be made to ensure a scalable end-to-end process.

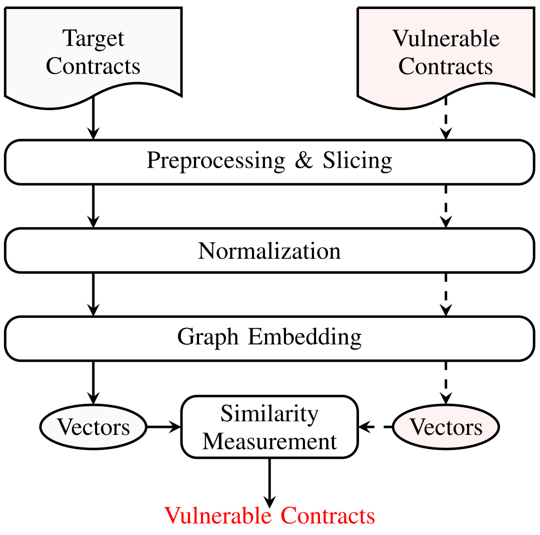


Figure 3.1: The entire process as laid out by the authors. Illustration taken directly from [HHY+21].

Figure 3.1 depicts the process pipeline of [HHY+21]. By manipulating the CFGs of smart contract bytecodes, the authors of [HHY+21] are able to extract graph-like datastructures called *slices*. The embedding framework *graph2vec* is used to learn the embedded representation of these slices in an unsupervised manner. These steps are performed using different datasets, one containing contracts from the blockchain and the other containing known-to-be vulnerable contracts. Finally, they identify slices that embody the most of a vulnerability's logic, which they then match against slices extracted from all other contracts. Since the embedded representations of slices are easily comparable, they are able to discover previously undetected vulnerabilities within real smart contracts.

We try to re-enact their steps, and extend their approach by the process pipeline depicted in Figure 3.2. The most significant differences are related to optimizations that were made in order to increase contract throughput: We operate on a smaller dataset by grouping together semantically equivalent contracts, and we leverage the immense parallelization capabilities of the GPU compared to the CPU for the calculation of similarities. For the sake of modularity in our experiments we consolidate the results of each stage in individual files in the JavaScript Object Notation (JSON) format.

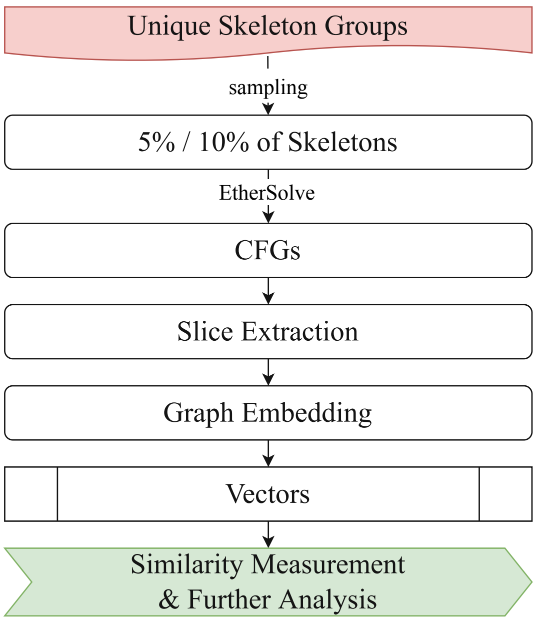


Figure 3.2: Our application of the process. Each step's resulting data is stored in a separate file to enable checkpoint-like behaviour when running experiments with different parameters/tunings etc.

3.1 Obtaining Bytecodes

The first step of our process is to obtain the bytecodes of smart contracts. Since the size of the body of contracts has a large impact on the runtime of the whole process, it is crucial to reduce this number to a minimum. Thankfully, it is no secret that a large portion of smart contracts are functional clones of each other, with little to no semantic differences [dAS20b]. At the time of writing, there are 490 k unique smart contracts among the 47.6 M deployed ones on the Ethereum main chain. In order to further reduce the number of unique contracts, and group those together whose code differ only minimally, we generate smart contract skeletons as described in [dAS19b, dAS20d] and section 2.6.

3.1.1 Smart Contract Skeletons

By applying a series of transformations to contract codes, we are able to lower the number of truly *unique* contracts even further. These transformations comprise the replacement of three code sections by zeros: The meta-data, constructor arguments and the PUSH arguments that are read off of the bytecode sequence. According to [dAS19b, dAS20d, dAS20b], these modifications have little to no impact on the functional behaviour of contracts, and allow us to "*transfer knowledge gained for one contract to others with the same skeleton*" [dAS20b]. Thus, by assessing the similarity of two contracts that are representative for a skeleton group, we are able to reason about the similarity of two entire groups of contracts sharing the same skeletons.

Using a relational database, we are able to extract contract bytecodes — one representing each skeleton group. The reduction of 489 k unique contracts to roughly 234 k unique *skeletons* yields benefits through the entire process pipeline and reduces the runtime by more than half compared to when operating on unique contracts alone.

3.2 Generating the Control Flow Graphs

The current and the following section deviate from [HHY+21]. In order to extract slices from contract bytecodes, their CFGs needs to be computed beforehand. We base our approach to generate CFGs on [CCCP21]. This is done due to the fact that the authors of [HHY+21] have not gone into detail on how they create the CFGs. Instead, they refer to two existing methods described in [LCO+16] and [TDC+18]. We assume that the CFG-generation and slicing procedure were done simultaneously, though the evidence for this is vague. For these reasons, we opted to use an external tool called EtherSolve [CCCP21][1] for CFG generation instead.

The generation of the CFGs is agnostic of the source the contract bytecodes stem from. This allows for combining contracts extracted from the real ethereum blockchain with synthetically compiled datasets as we will do in our later experiments using a modified

[1]https://github.com/SeUniVr/EtherSolve

version of EtherSolve. The modifications we made comprise functionalities for the processing of multiple contracts at once, rather than having to start a new process for each individual bytecode. We also utilize our system's multiprocessing capabilities by running multiple bytecodes in parallel. The resulting CFG data is combined within a single output file in the JSON format.

Due to different possible stack configurations, EtherSolve's results may include edge lists containing the same directed edge more than once. We make the simplification to reduce these duplicated edges to a single one in an attempt to limit the runtime in our slicing stage.

An example for a CFG was already shown in Figure 2.3. While EtherSolve's outputs contain lots of additional information on a smart contract, we make use of only a few fields of the generated JSON object while extracting slices in the next steps. In particular, we care about the nodes and the edges within the CFG, and discard the remaining metadata. The output file at this stage consists of a single JSON dictionary, containing contract addresses as keys and EtherSolve's analysis data as value. Since we anticipated experiments with files of up to 50 GB in size, we make use of an iterative JSON parser[2] that is able to stream the json key-value pairs, rather than having to load the JSON object into the memory before being able to further process it. The remaining processes in the pipeline have been optimized in a similar fashion.

3.3 Preprocessing & Slicing

This section explains how we perform the slicing procedure. We have tried to stay as true to [HHY+21] as possible, but due to lack of details we had to make numerous assumptions when faced with uncertainties. We also had to undertake measures for optimizing the runtime, as our interpretation of the original method would have been infeasible to compute.

The rationale behind the slicing procedure in [HHY+21] is to remove code sections that do not carry relevant information or code semantics with them. As their work had an emphasis on security, they decided that they would only be interested in segments of code in which user provided data, or data that can be influenced by an external entity, is handled. For this, they have defined a set of opcodes called *slicing criteria*. These consist of EVM instructions which rely on, or handle information stemming from outside the contract's own program logic, e.g. **CALLDATALOAD** or **BLOCKHASH**. Such instructions mark the beginning of a slice.

As visible in Table 3.1, Huang et al. defined 4 categories for such criteria. During our experiments we found that the matching potential of contracts was dependent on how many slices we extract per contract. In an attempt to raise the average number of slices, we extended these operations by adding other instructions we deemed fitting to the

[2]https://github.com/ICRAR/ijson

Category	**Operation**
Transaction Data	CALLDATALOAD, CALLER, CALLVALUE, **CALLDATASIZE**
Block Data	BLOCKHASH, TIMESTAMP, **NUMBER**, **DIFFICULTY**, **GASLIMIT**
Storage Data	SLOAD
Return Values	CALL, **CALLCODE, DELEGATECALL, STATICCALL, CREATE, CREATE2**
Miscellaneous	**SHA3**, **CODESIZE**, **EXTCODESIZE**, **EXTCODEHASH**, **CHAINID**, **SELFBALANCE**, **PC**, **GAS**

Table 3.1: Slicing criteria extracted from the original paper. Instructions and categories in **bold** were added by us.

existing categories, and added another category for instructions that we thought would benefit our cause.

The goal of the slicing procedure is to trace in- and outputs of instructions that are picking up on the execution results of the current *slicing criterion* — an approach that is also referred to as *taint analysis* [TPF+09]. In order to follow said arguments and outputs across the entire simulated execution of a contract, we need another mapping between opcodes and so-called *tags*. This mapping serves to reduce the number of possible categories for operations, taking away variability to some degree. Operations with related semantics (e.g. **CALLDATALOAD**, **CALLDATASIZE**) are assigned to the same group and are given a group tag (as described by the original paper) as well as a group symbol (as used by us). Again, we decided to extend instructions that were defined in [HHY+21] in a manner that seemed meaningful for us, especially due to the fact that [HHY+21] explicitly mentions the incompleteness of the table they provide. Our **additions** to their defined groups and tags can be seen in Table 3.2.

A core concept of the original method is the concatenation of input arguments with the group tag (or symbol in our case) of the currently executed instruction. Elements inside the stack are represented by strings. During the execution, these strings may be concatenated to each other to form new, arbitrarily-sized stack elements. We are able to decrease our memory footprint by using single-lettered group symbols in favor of the original tags introduced by [HHY+21]. Since the graph embedding stage will build upon tokenized labels of nodes, this simplification will not affect the results in any way.

Instructions	Group Tag	Group Symbol (Instruction Output)
CALLDATALOAD, CALLDATACOPY[1], **CALLDATASIZE**	calldata	a
CALLER, ORIGIN, **CALLVALUE**, **ADDRESS**, **BALANCE**, **GASPRICE**	tx_data	b
BLOCKHASH, TIMESTAMP, **COINBASE**, **NUMBER**, **DIFFICULTY**, **GASLIMIT**, **BASEFEE**, **CHAINID**, **SELFBALANCE**, **PC**, **GAS**	blk_data	c
SLOAD, **SHA3**	sto_data	d
CALL, **RETURNDATASIZE**, **CALLCODE**, **DELEGATECALL**, **CREATE**, **CREATE2**, **STATICCALL**, **CODESIZE**, **EXTCODESIZE** **EXTCODEHASH**	call_res	e
ADD, MUL, **SUB**, **DIV**, **SDIV**, **MOD**, **SMOD**, **ADDMOD**, **MULMOD**, **EXP**	arith_res	f
NOT, AND, OR, **XOR**, **SHL**, **SHR**, **SAR**	bit_res	g
LT, GT, SLT, EQ, **SGT**, **ISZERO**	cmp_res	h
MLOAD, **MSIZE**	mem_data	i
PUSH[2]	literal	j

[1] While this instruction is included in the table found in [HHY+21], it does not produce any output onto the stack that could be tagged. We are including it here for the sake of completeness but leave it out in the actual code.
[2] Includes all operations of the PUSH family.

Table 3.2: Mapping between instructions and their output tags. Original table taken from [HHY+21]. Instructions written in **bold** were added by us.

3.3.1 Simulating Bytecode Execution

At this point we have to perform the simulated execution of a given bytecode within a CFG to extract the slices. Our focus of interest during execution lies in the opcode that is currently being executed, as well as the contents of the stack at that time. To this end, we deploy a simplified stack machine, and use it to extract the data of interest.

The core element of this process is the stack. It keeps track of arguments that are used for instructions as well as their outputs. Moreover, it allows us to track the usage of data introduced by a criterion across the entire line of execution and recognize its usage by subsequent operations up until the point at which every stack element that is derived from the criterion is eventually consumed.

We had to make a compromise between having an accurate stack representation, and our process running in feasible time. The most accurate stack representation at start time of slicing would be obtained by simulating execution beforehand, i.e. by having an *outer* recursion traversing the graph. However, we realized in our experiments that this option is computationally infeasible for many of our CFGs, and opted against using this variant. Instead, we pass a symbolic stack configuration of 32 elements to the subroutine traversing the subgraph. See subsection 3.7.1 for details.

Our starting point is the collection of nodes from the CFG as depicted in Figure 2.3. We iterate over all nodes in the graph in no particular order. Once a criterion (as defined in Table 3.1) is met, we start the slicing procedure as laid out in Figure 3.3 by passing the remaining subgraph, an instance of the static stack configuration, and an empty list for storing the extracted slices, to a subroutine. Each instruction is then processed one at a time while recursively traversing over all paths between the initiating criterion and the CFG's exit block. At the same time we simulate the effects of each instruction on the stack by *pushing* and *popping* the correct amount of arguments from and to the stack respectively.

Before branching out into the subgraph, however, the criterion's group symbol is pushed onto the current stack configuration as an upper case, single-lettered string, which we will refer to as **slice key** from now on. This is done in order to distinguish between instruction outputs that are of the same category as the slicing criterion — we only want to trace the *exact same* stack element that was originally introduced by the criterion. In order to perform an accurate execution, we create a ruleset for handling different cases based on which opcode is currently encountered

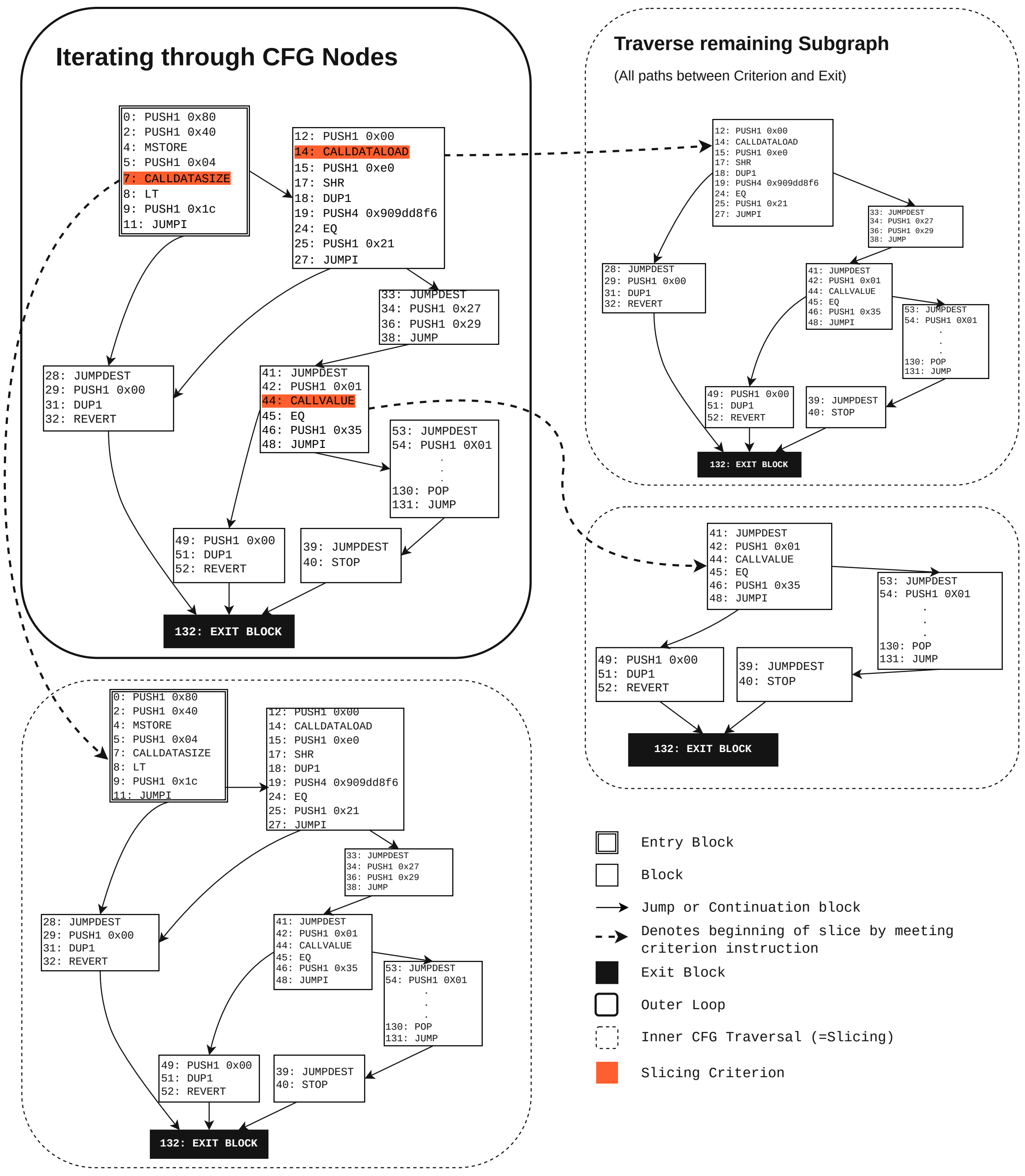


Figure 3.3: Recursive algorithm for traversing the CFG. The outer loop, while iterating over CFG nodes, initiates slicing once a criterion is met. Illustration adapted from [CCCP21].

Ruleset I - Execution and Extraction

PUSHN The PUSH group's symbol ('j' in this case) is pushed onto the stack. N determines the offset to the next operation read from the bytecode sequence.

SWAPN Swaps the topmost element in the stack with the N^{th} element in the stack.

DUPN Duplicates the N^{th} topmost element and pushes it onto the stack.

The remaining instructions are executed by correctly handling the number of arguments produced and consumed. To this end, we keep a mapping $\boldsymbol{m} : o \rightarrow (\delta, \alpha)$, where o denotes an instruction, δ the amount of arguments consumed, and α the amount of arguments produced from and onto the stack by o respectively. In the case that the instruction produces an output (i.e. $\alpha > 0$), we concatenate the δ popped arguments to each other, prepend the group symbol of o and push the resulting string back onto the stack. If no output is produced, the popped arguments are simply discarded.

If the **slice key** is encountered in the executing instruction's arguments, and the executing instruction is neither of the **PUSH**, **SWAP** or **DUP** families, we append the instruction along with its arguments to our *slices* collection. Additionally, in case we encounter an arithmetic or a bitwise operator, we employ a normalization technique described in section 3.3.1 below.

Since we are dealing with graphs, we implement the subroutine as a recursive function that will branch once a junction (i.e. a **JUMPI**) is met. This way we make sure that all paths (within a certain recursion depth) are traversed, and the slices correctly extracted. We apply **Ruleset I** to every instruction in the executing subgraph during slicing, and finish slicing only in one of the following cases:

Ruleset II - End of Slice

INVALID If the execution encounters an invalid opcode (i.e. an undefined byte-instruction), we extract the mnemonic **INVALID** and end the simulation for the current branch.

REVERT, RETURN The authors of [HHY+21] reason that these instructions are of interest since they indicate (abnormal) terminations of a transaction. Thus, when encountered, either of these operations are included into the slice and the simulation ends for the current branch.

Slice Key not in Stack Once the last element in the stack containing the slice key is consumed, we add the slice to a collection and end the simulation for the current branch.

The extracted slices are appended to a global list before returning from the current branch. This results in one contract having ~128 slices on average.

Instruction Normalization

To remove even more variability introduced by compilers, the authors of [HHY$^+$21] suggest a normalization technique for arithmetic and bitwise operations *before* extracting them into a slice. To this end, we reorder arguments alphabetically for the commutative instructions **ADD**, **MUL**, **AND**, **OR**, **XOR**, **EQ** (e.g. $add(b, a) = add(a, b)$). The same applies to the operations **LT**, **GT**, **SLT**, **SGT**, with the exception that *iff* reordering occurs, we also replace the instruction with its semantic opposite (e.g. $lt(b, a) = gt(a, b)$). This ensures that the instruction's original semantics are kept.

3.3.2 Slicing Example

Using the entry node in Figure 3.3 as reference point and Figure 3.4 as visualization, we continue to provide an example.

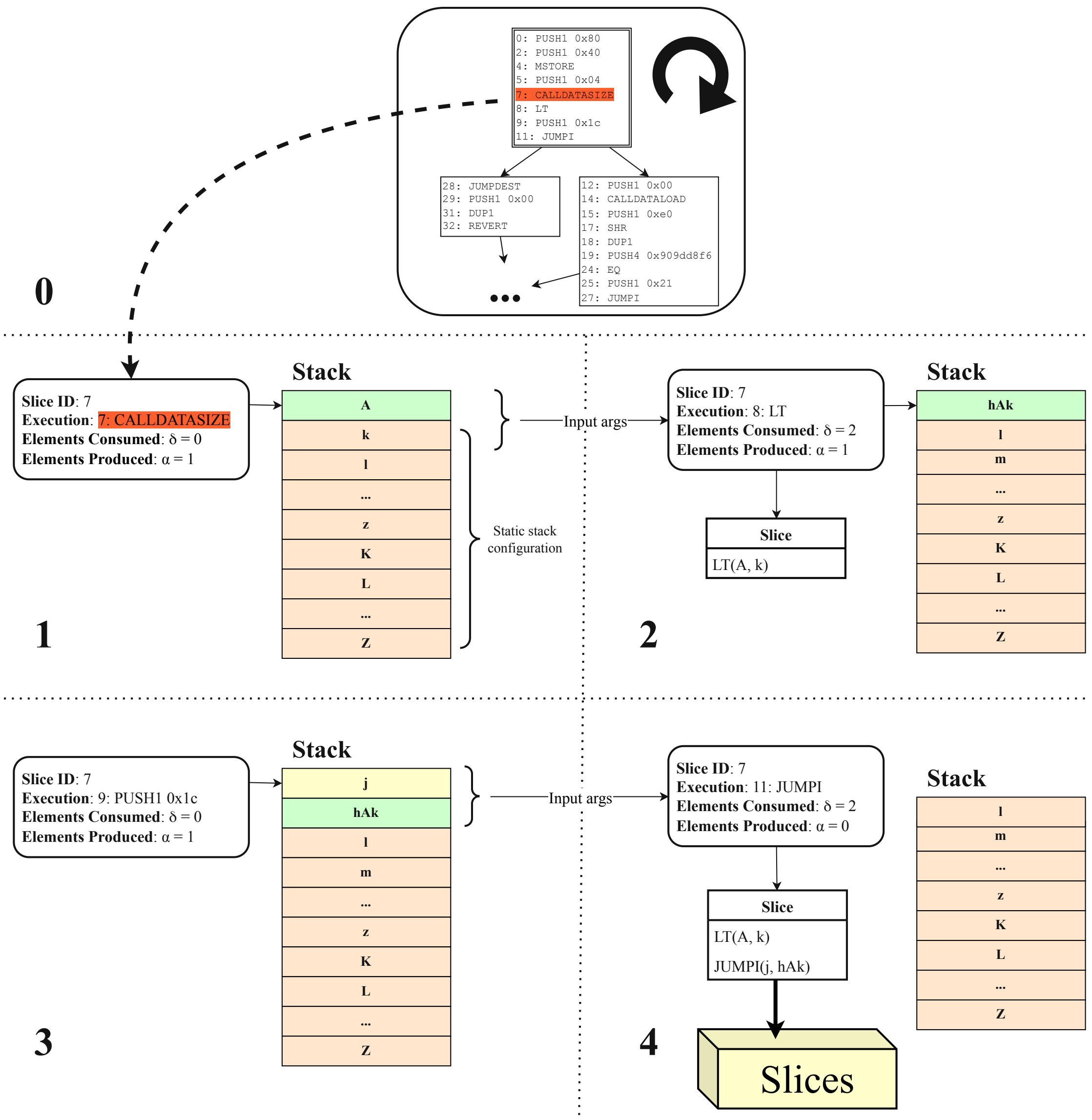
Step 0 The outer loop iterates over the bytecodes of all nodes until it detects one of the criteria defined in Table 3.1.

Step 1 The criterion **CALLDATALOAD** is encountered. Slicing begins with a static stack configuration containing dummy values (i.e. values that are not assigned to criteria groups as symbols). The criterion introduces the slice key, i.e. its capitalized group tag, onto the stack.

Step 2 Instruction **LT** is encountered. It consumes two elements from the stack. Since one of them contains the slice key (**A** in this case), we extract the instruction along with its arguments as part of the slice. The instruction's output is obtained by concatenating its arguments and prepending **LT**'s group tag. Note that instruction reordering is not performed here due to the first argument, **A**, being alphabetically smaller than the second argument, **k**.

Step 3 The next instruction is of the **PUSH** family. We push an element, **j**, onto the stack while consuming none.

Step 4 Lastly, a **JUMPI** is encountered. This argument consumes the two topmost elements, and thus, is extracted as part of the slice as well. Since it produces no output, the slice key is no longer present in the stack after its execution, which indicates that the end of this slice has been reached. Lastly we append the slice to a list of slices, return from the current subroutine, and continue iterating over the nodes in **Step 0**.



```
0: PUSH1 0x80
2: PUSH1 0x40
4: MSTORE
5: PUSH1 0x04
7: CALLDATASIZE
8: LT
9: PUSH1 0x1c
11: JUMPI
```

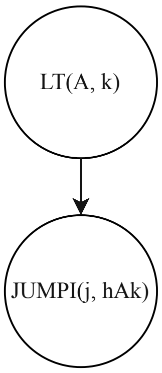
```
28: JUMPDEST
29: PUSH1 0x00
31: DUP1
32: REVERT
```

```
12: PUSH1 0x00
14: CALLDATALOAD
15: PUSH1 0xe0
17: SHR
18: DUP1
19: PUSH4 0x909dd8f6
24: EQ
25: PUSH1 0x21
27: JUMPI
```

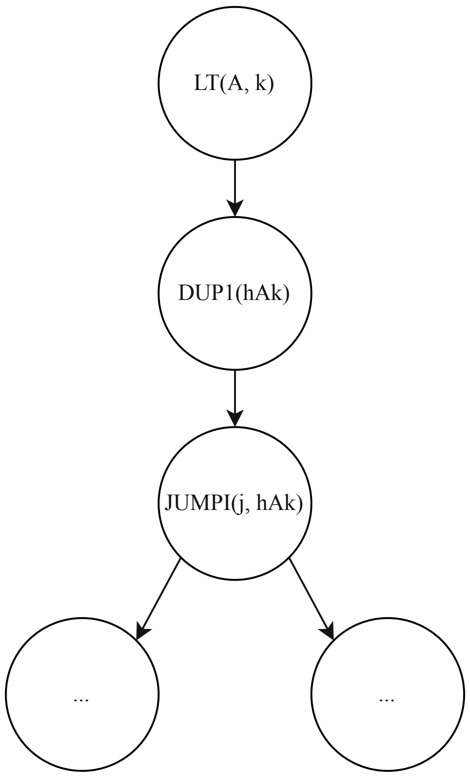
🟧 Dummy value as part of the static stack
🟩 An element containing the criterion's tag
🟨 Remaining stack elements
🟥 Slicing criterion

Figure 3.4: Step-by-step example based on the first criterion found in the entry block of the CFG in Figure 3.3.

In [HHY+21, p. 2148], it is clearly stated that in this procedure each instruction is viewed as a node within a slice, and that each slice is viewed as a graph. Many of the extracted slices contain graphs where each node has exactly one predecessor and one successor (e.g. the slice extracted in Figure 3.4). This is due to the fact that the slice key is often consumed before a **JUMPI** to another block occurs. For other slices, we are able to retain the CFG's structure by introducing edges into the extracted slice whenever a junction is met. Figure 3.5 draws a comparison between the two kinds of graphs.



(a) The graph for the slice extracted in Figure 3.4. There is no junction because the slicing ended immediately after a **JUMPI** was encountered.



(b) Another example of a similar slice. Here the argument of **JUMPI** is duplicated before being consumed. The criterion's symbol is still present in the stack so both paths continue to be executed afterwards.

Figure 3.5: Two example slices/graphs.

3.4 Graph Embedding

Once the contract slices are obtained, we continue with the graph embedding. We make use of a graph embedding library, namely *graph2vec*[3]. As was the case in previous sections, we modify this library to allow for data-streaming from files. This unfortunately comes with the drawback that learning takes longer due to bandwidth constraints which is why we only make use of this modification when memory becomes a constraint, as is the case in one of our experiments.

We call graph2vec with all default parameters except for the "dimensions" parameter, which we set to 64 as the authors of [HHY$^+$21] do. This configuration is used across all experiments in an effort to try and limit the already large number parameters in our experiment variations. As input file for graph2vec we provide the result of the slicing stage. This results in a CSV file where each row corresponds to a single slice of a contract, containing the contract's skeleton, address, slice id, and the 64-dimensional embedding vector. Table 3.3 exemplifies the data produced in this step. We may use the terms *slice* and *vector* interchangeably in the following sections.

	Slices																
Dim.	**0**	**1**		**j-2**	**j-1**	**j**	**j+1**		**k-2**	**k-1**		**l**	**l+1**		**m-2**	**m-1**
0	0.97	0.88		-0.82	-0.08	0.53	-0.09		0.73	0.37		-0.84	-0.57		-0.98	-0.29
1	0.08	0.53		-0.88	0.41	0.14	-0.91		0.08	-0.56		-0.67	-0.62		0.91	-0.74
2	-0.33	-0.04		0.50	0.94	0.61	-0.34		0.58	-0.68		0.26	-0.88		-0.94	-0.71
3	0.56	0.74	…	-0.51	0.24	-0.75	-0.54	…	-0.03	0.42		0.33	-0.95	…	0.08	0.97
⋮	⋮			⋮		⋮			⋮		…	⋮			⋮	
60	0.65	-0.56		-0.56	-0.07	-0.83	-0.45		-0.84	0.24		-0.40	0.70		-0.76	-0.94
61	-0.26	-0.43		-0.05	-0.62	0.19	0.56		-0.20	0.65		-0.80	-0.89		-0.35	-0.61
62	0.19	-0.67		0.23	0.16	0.29	-0.89		0.83	0.39		-0.75	0.87		0.73	0.92
63	0.95	0.94		0.53	-0.69	-0.73	-0.48		-0.20	0.84		-0.06	0.95		0.04	-0.78
	Contract 1					**Contract 2**						**Contract N**				

Table 3.3: Result of the graph embedding. This table is a transposed depiction of the actual CSV file. Each slice corresponds to one 64-dimensional vector in this array.

3.5 Similarity Measurement

As is usual when dealing with embedding vectors [MCCD13], the authors of [HHY$^+$21] calculate the similarity between two slices by using the *cosine similarity*:

$$\cos(\theta) = \frac{a \cdot b}{\|a\| \, \|b\|} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n}(a_i^2)} \cdot \sqrt{\sum_{i=1}^{n}(b_i^2)}} \tag{3.1}$$

The result of the cosine similarity function is a real value between -1 and 1, and is commonly used to measure the angle between any two vectors. In our 64-dimensional

[3]https://github.com/benedekrozemberczki/graph2vec/

vector space, however, this value is regarded as the *similarity* between two vectors, with a cosine of -1 meaning two slices are exact opposites of each other, and a cosine of 1 meaning their vectors are congruent.

3.5.1 1-to-1 Similarity

Due to the focus of [HHY+21] lying on detecting reused code vulnerabilities, their matching logic follows a *1-to-1* similarity calculation: For every vulnerable contract they identify the *target slice* that encapsulates the logic of the vulnerability the most, and match this slice against all other slices using Equation 3.1 [HHY+21, p. 2149]. To recreate the results of their *1-to-1* matching, we craft our own data-set of slices containing contract vulnerabilities, and try to conduct experiments in the same fashion. However, the problem quickly grows in complexity when trying to relate entire contracts — each potentially spanning hundreds of slices — to one another.

3.5.2 N-to-M Similarity

The problem of *n-to-m* slice matching is a variation of the classical Assignment problem [10.63]. Instances of this problem are commonly represented using one of the two following data-structures:

Matrix Representation Consider a two-dimensional matching matrix of size $n \times m$ between two contracts c_1 and c_2. Then, v_{ij} for $i < n$ and $j < m$ denotes the matching value between slice i of contract c_1 and slice j of contract c_2. Reference implementations for the problem using this variant exist. A common textbook algorithm for this representation is the *Hungarian algorithm* [MtSD07].

Bipartite Graph Representation Due to each node in either partition having a bidirectional edge to every node in the other partition, we obtain two fully connected partitions in a bipartite graph. This peculiarity makes the memory consumption of both representations roughly the same. Previous works describe linear-runtime approximation algorithms given any fixed error bound [DP14]. However, neither did the authors of [DP14] implement this algorithm, nor were we able to find an implementation online.

In our experiments, we opt for the first representation variant, and enjoy the advantage of having a readily implemented reference algorithm for finding an optimal solution. We further help ourselves by exploiting the fact that the cosine similarity between two vectors can be easily computed in batch using matrix multiplication — an operation that is notoriously well optimized for running on GPUs. Before doing so, however, we further simplify the previously obtained vector array (see Table 3.3) by transforming each of the slicing vectors into their corresponding unit vector. This step equates the denominator in Equation 3.1 to $\mathbf{1}$, conveniently reducing the right part of the equation to $\sum_{i=1}^{n} a_i \cdot b_i$. In succeeding sections we will refer to the unit vectors array as $\hat{\mathbf{V}}$.

We iterate over all indices that delimit the individual contracts, and perform the similarity calculation between a single contract $\mathbf{c_i}$ and all other contracts one at a time. To this end, we pass $\mathbf{\hat{V}}$ and $\mathbf{c_i}$ as arguments into TensorFlow's *matmul*[4] function, and indicate that $\mathbf{c_i}$ shall be transposed before execution via a third flag argument, *transpose_b*.

We summarize the matrix dimensions below and follow with an example:

$$\underset{p_i \times q}{S_i} = \underset{p_i \times 64}{c_i^{\intercal}} \cdot \underset{64 \times q}{\hat{V}} \tag{3.2}$$

where

$\mathbf{S_i}$: similarity matrix for contract c_i
$\mathbf{\hat{V}}$: unit vectors of all slices where each column corresponds to one slice
$\mathbf{c_i}$: submatrix containing the slices/vectors that make up the i^{th} contract
$\mathbf{c}_i^{\intercal}$: transposition of submatrix c_i
$\mathbf{p_i}$: number of slices of contract c_i
$\mathbf{q}$: total number of slices

Example: Similarity Calculation for Contract $\mathbf{c_1}$

$$\mathbf{S_1} = \underbrace{\begin{bmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,62} & v_{0,63} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ v_{j,0} & v_{j,1} & \cdots & v_{j,62} & v_{j,63} \end{bmatrix}}_{\mathbf{c}_1^{\intercal}} \cdot \underbrace{\left[\begin{array}{ccccccccccc} \multicolumn{3}{c}{\overbrace{\hphantom{v_{0,0} \cdots v_{0,j}}}^{\mathbf{c}_1}} & \multicolumn{3}{c}{\overbrace{\hphantom{v_{0,j+1} \cdots v_{0,k}}}^{\mathbf{c}_2}} & \cdots & \multicolumn{3}{c}{\overbrace{\hphantom{v_{0,l} \cdots v_{0,m}}}^{\mathbf{c}_n}} \\ v_{0,0} & \cdots & v_{0,j} & v_{0,j+1} & \cdots & v_{0,k} & \cdots & v_{0,l} & \cdots & v_{0,m} \\ v_{1,0} & \cdots & v_{1,j} & v_{1,j+1} & \cdots & v_{1,k} & \cdots & v_{1,l} & \cdots & v_{1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ v_{62,0} & \cdots & v_{62,j} & v_{62,j+1} & \cdots & v_{62,k} & \cdots & v_{62,l} & \cdots & v_{62,m} \\ v_{63,0} & \cdots & v_{63,j} & v_{63,j+1} & \cdots & v_{63,k} & \cdots & v_{63,l} & \cdots & v_{63,m} \end{array}\right]}_{\mathbf{\hat{V}}} =$$

$$= \underbrace{\left[\begin{array}{cccccccccc} \multicolumn{3}{c}{\overbrace{\hphantom{1 \cdots s_{0,j}}}^{\mathbf{c}_1 \cdot \mathbf{c}_1^{\intercal}}} & \multicolumn{3}{c}{\overbrace{\hphantom{s_{0,j+1} \cdots s_{0,k}}}^{\mathbf{c}_2 \cdot \mathbf{c}_1^{\intercal}}} & \cdots & \multicolumn{3}{c}{\overbrace{\hphantom{s_{0,l} \cdots s_{0,m}}}^{\mathbf{c}_n \cdot \mathbf{c}_1^{\intercal}}} \\ 1 & \cdots & s_{0,j} & s_{0,j+1} & \cdots & s_{0,k} & \cdots & s_{0,l} & \cdots & s_{0,m} \\ s_{1,0} & \cdots & s_{1,j} & s_{1,j+1} & \cdots & s_{1,k} & \cdots & s_{1,l} & \cdots & s_{1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{j,0} & \cdots & 1 & s_{j,j+1} & \cdots & s_{j,k} & \cdots & s_{j,l} & \cdots & s_{j,m} \end{array}\right]}_{\mathbf{S_1}} \tag{3.3}$$

where

$\mathbf{v_{i,j}}$: j^{th} value of vector i / slice i
$\mathbf{s_{i,j}}$: cosine-similarity between slices i and j

[4] *matmul* in the official TensorFlow documentations

We proceed to conduct experiments using two different ways for calculating *m-to-n* similarities, one finding an optimal solution and the other using a heuristic method.

Variant I: Optimal Slice Matching

Since the speed of the similarity computation on the GPU and the speed at which the result can be further processed on the CPU are quite asymmetric, we implement the best-match finding algorithm using multiple worker processes, each running on its own CPU core. That means, each result array $\mathbf{S_i}$ is passed to a worker process that will again iterate over all contract indices, and for each sub-result $\mathbf{c}_j \cdot \mathbf{c}_i^{\intercal}$ apply the linear sum assignment algorithm. Using this parallelization, we can somewhat compensate for the CPU's slow performance when compared to the GPU, but are still able to detect idle time of the latter.

		Contract N					
		Slice 1	**Slice 2**	**Slice 3**	**Slice 4**	**Slice 5**	**Max**
Contract M	**Slice 1**	-0.857	0.527	-0.972	-0.144	0.122	**0.527**
	Slice 2	0.805	-0.968	-0.805	0.022	**0.798**	**0.805**
	Slice 3	-0.250	0.313	-0.821	0.066	0.113	**0.313**
	Slice 4	0.094	0.034	0.420	-0.692	-0.846	**0.420**
	Slice 5	-0.462	-0.392	0.671	**0.586**	0.922	**0.922**
	Slice 6	0.281	0.437	-0.647	-0.699	-0.480	**0.437**
	Slice 7	-0.740	0.602	**0.824**	0.017	0.476	**0.824**
	Slice 8	0.547	0.192	0.590	-0.498	-0.276	**0.590**
	Slice 9	-0.562	**0.615**	-0.302	-0.594	-0.845	**0.615**
	Slice 10	-0.037	-0.346	-0.828	-0.235	0.040	**0.040**
	Slice 11	-0.341	0.478	0.441	-0.832	0.313	**0.478**
	Slice 12	**0.645**	-0.705	-0.116	-0.578	-0.637	**0.645**
	Max	**0.805**	**0.615**	**0.824**	**0.586**	**0.922**	

Table 3.4: An example for a matching matrix between slices of two example contracts N and M. Optimal match indicated in **bold**. We include the maximum value of each column/row to stress the fact that the optimal solution cannot generally be found by a greedy algorithm, i.e. by picking the largest values of the column/row.

Worker processes make use of a reference implementation from the scipy library[5] which is based on the Jonker-Volgenant algorithm, but has been modified to have no initialization. It has an upper bound runtime of $\mathbf{O(n^3)}$ [Cro16]. Table 3.4 shows an example of how slices of two different contracts are matched, with the resulting match being of size $min(|c_n|, |c_m|) = 5$. We define a threshold $\mathbf{t} = \mathbf{0.97}$, and regard contract pairs with an average matching value larger or equal to $\mathbf{t}$ as *similar*. These contracts, combined with additional metadata, are extracted into a separate file for further analysis.

[5]Linear sum assignment problem in the official scipy documentations

Initial experiments were conducted using a sample size of 5 % of skeleton-contracts, but we quickly realized the need for an even more efficient process. Thus, we decided to explore further possibilities to push computation towards the GPU.

Variant II: Heuristic Slice Matching

Albeit not without sacrificing the optimality of our solution, we try to extract similar information regarding contract matches using other functionalities provided by the TensorFlow library. The functions *tf.math.segment_max* and the ones of the *tf.math.reduce_** family have turned out to be particularly useful for our intentions.

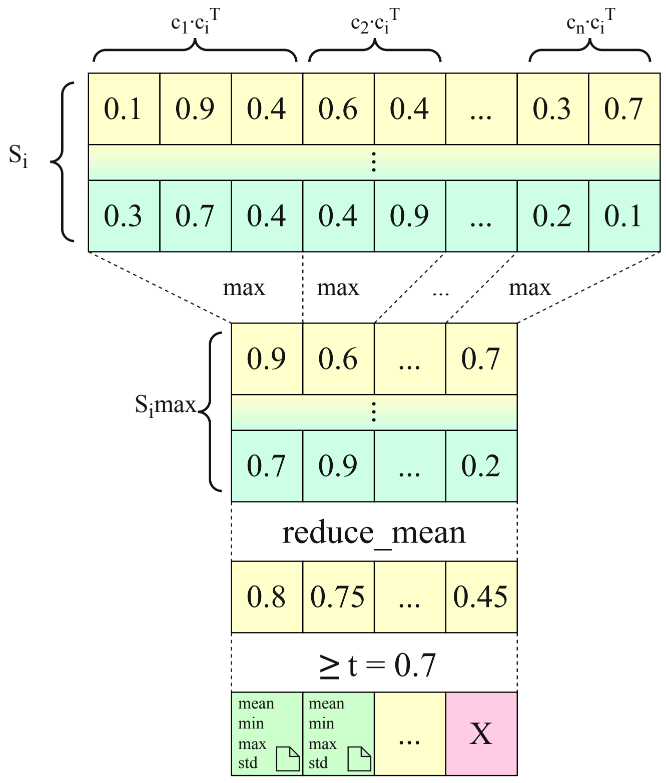


Figure 3.6: Illustration of the *tf.math.segment_max* function. For demonstration purposes we display $\mathbf{t} = \mathbf{0.7}$. Actual values used in our experiments range from 0.85 to 0.95. Figure adapted from the official TensorFlow documentation [6].

Picking up on the matching result $\mathbf{S_i}$ of Equation 3.3, Figure 3.6 depicts an example of a $\mathbf{p_i} \times \mathbf{q}$-dimensional tensor being reduced to a $\mathbf{p_i} \times \mathbf{r}$-dimensional tensor, where $\mathbf{r}$ denotes the total number of contracts in our experiment. We denote the resulting matrix with $\mathbf{S_i max}$, and use it to perform further aggregations on a per-contract basis. Initially, we calculate only the mean over individual contracts in $S_i max$ using *reduce_mean* and

[6]tf.math.segment_max - Official TensorFlow documentations

define another threshold **t** to determine whether a contract qualifies for extraction or not. Contracts that do not qualify are dismissed, while to the others we apply further reductions. In particular, we calculate the minima, maxima and the standard deviations for each qualifying contract in **$\mathbf{S_i}$max**, and extract these metrics to a file.

In addition to the aforementioned aggregations, we make use of TensorFlow's *AutoGraph*[7] module, which converts functions written in python code into execution graphs for the GPU. This dramatically reduces communication overhead between CPU and GPU inbetween execution cycles, and results in immensely improved data throughput. With these additions, we are able to fully exploit the GPU's capacities.

To provide context, one of our earlier experiments using a 5 % sample size of skeletons, and an average slice count of ~80 slices per contract would take several hours using **Variant I**, while the method described in **Variant II** would finish in less than two minutes. Unsurprisingly, this effect is amplified when running experiments with larger sample sizes.

3.6 Hypotheses

In contrast to the optimal method, our heuristic approach produces a pair of each metric for both directions of a match. We continue to denote $S_i max$ for contract j and $S_j max$ for contract i with $m_{i,j}$ and $m_{j,i}$ respectively. The metrics include the means, minima, maxima and standard deviations of $m_{i,j}$ and $m_{j,i}$. As we are mostly focused on the mean of these values, we introduce the notions $\overline{m}_{i,j}$ to denote the mean of $m_{i,j}$, and $\overline{m}_{j,i}$ to denote the mean of $m_{j,i}$. This allows us to define the symmetric similarity score $M_{i,j} = M_{j,i}$ between two contracts follows:

$$M_{i,j} = \min\left(\overline{m}_{i,j}, \overline{m}_{j,i}\right) \tag{3.4}$$

We reason that there are a few deterministically computable contract features that serve as useful metrics for determining how contracts relate to each other. First and foremost we assume that contracts featuring the same *function signatures* are likely to perform semantically related tasks. Thus, a plausible metric to compute a similarity score between two contracts is to calculate the *Jaccard index* over their sets of function signatures A and B respectively, defined as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.5}$$

The rationale behind this is that contract authors usually name functions in a predictable manner, either following existing conventions or the same literature. Further, we think that accounts belonging to the same entity are likely to deploy several versions of the

[7]tf.autograph - Official tensorflow documentation

same contract more than once, or at least that these accounts are likely to re-use code among the contracts they deploy.

Taking this idea further, one could argue for several such correlations, albeit with much less confidence: If two contracts are deployed within quick succession, they are more likely to be related than two contracts with deployments separated by millions of blocks. We want to investigate whether the *temporal locality* of the deployment of two contracts is a measurable indicator of their similarity. A final metric that we want to examine in the same way is the code length of a contract. It is not too far-fetched to believe that contracts comprising the same semantics hardly differ in their respective code lengths.

As is the case with the embedding network word2vec and its *skip-gram* model, we also suspect that a larger dataset for training a graph embedding network will have a positive impact on similarity matching. Combining these thoughts leads to the following hypotheses, ordered by how confident we are in making them.

Hypothesis 1: A higher Jaccard index J between two contracts i and j generally implies a higher similarity score $M_{i,j}$. A larger dataset is more likely to exhibit this behaviour.

Hypothesis 2: If two contracts have the same creator, they are more likely to be similar than two contracts with distinct creators. A larger dataset is more likely to exhibit this behaviour.

Hypothesis 3: Contracts with low block difference between their deployments are more likely to be similar. A larger dataset is more likely to exhibit this behaviour.

Hypothesis 4: Contracts with low difference in code length are more likely to be similar. A larger dataset is more likely to exhibit this behaviour.

We try to investigate these hypotheses, and by providing appropriate correlation metrics we want to argue either in favor of or against them.

3.7 Contributions

While we made an effort to highlight all deviations from the source material in the previous sections, we want to summarize our contributions in this section.

3.7.1 Stack Content before Slicing

Since starting the slicing procedure with an empty stack may lead to *IndexOutOfBoundsErrors* (namely, when the sequence succeeding the criterion consumes more arguments than are available on the stack), our initial and faithful implementation required the stack being accurately simulated before slicing had begun as well. This, however, had immense performance drawbacks — especially with contracts that have a CFG with a high average branching factor. Further, we recognized that attempting to limit the width of the branching would not sufficiently decrease runtime either for any branching factor larger than 1. Thus, we decided to make use of a constant stack configuration consisting of 32 elements that is used when the slicing procedure is initiated.

3.7.2 Similarity Calculation

The authors of [HHY+21] focus on detecting similarities between individual slices, whereas our main interests lie in additionally detecting similarities across entire contracts. Due to the sheer amount of slices, we implement a resource-efficient comparison method using the GPU's parallelization capabilities for both kinds of matching, *1-to-1* and *N-to-M*.

3.7.3 Adjustments to External Tools

EtherSolve

As already mentioned, our additions to EtherSolve comprise functionalities to process contracts in batch, rather than one at a time. By doing so, contracts no longer need to be stored in individual files, and only one process needs to be launched instead of creating one for each contract. Furthermore, we add multiprocessing capabilities to EtherSolve, leveraging other CPU cores that would stay idle in the standard implementation. The resulting CFGs are stored in a single output file in the JSON format.

graph2vec

The implementation of *graph2vec* that we use will load all graphs into memory by default, before creating the embedding. For large corpora of contracts (i.e. the full set of skeleton-contracts), we were quickly limited by our system's memory. Feeling the need for a more memory-efficient process, we added an option to stream the graphs from file, rather than having to load them into memory first. Depending on the number of *epochs* (10 in our case), this comes with the drawback that creating the graph embeddings will take *much* longer — a penalty we found acceptable considering the lack of alternatives.

CHAPTER 4

Evaluation

In this chapter we lay out in detail the experiments that we conduct, as well as the steps needed in order to do so. We begin by providing information necessary to reproduce our experiments.

4.1 Data Recency

The data that we use for our evaluation is almost exclusively derived from the Ethereum mainnet blockchain. Its recency can be viewed in Table 4.1. In our experiments we operate on three different datasets of skeleton-contracts: A **full** dataset comprising all skeletons as well as two samples sized at **5 %** and **10 %** respectively. The dimensions of these datasets are provided in Table 4.3.

Latest Block	**13 599 999** (created Nov-12-2021)
Total number of deployed contracts	47 615 589
Contracts with distinct bytecode	488 883
Amount of distinct contract skeletons	233 533

Table 4.1: The recency and size of the data used in this work.

4.2 Test Unit Specification

All experiments during the course of this work were conducted on a system with an AMD Ryzen 3600–6C/12T CPU, an Nvidia RTX 2060 with 6GB GDDR6 and 32GB of DDR4 RAM, clocked at 3000 MHz. Table 4.2 depicts the various tools, programming languages and frameworks that were used during the course of this thesis.

Name	Category	Version	Used for	Link
Ubuntu	OS	21.10	-	-
Nvidia GPU Driver	System Driver	495.29.05	TensorFlow Dependency	-
Nvidia CUDA	GPU APIs	11.5	TensorFlow Dependency	-
Python	Programming Language	3.9.7	Implementation	official source
TensorFlow	ML Framework	2.7.0	Similarity Calculation	PyPI
EtherSolve	External Tool	0840e9d[1]	CFG generation	GitHub
graph2vec	External Tool	efa6157[1]	Graph Embedding	GitHub
numpy	Library	1.21.2	Array Manipulation Similarity Calculation	PyPI
SciPy	Library	1.7.1	Similarity Calculation Optimal Solution	PyPI
matplotlib	Library	3.5.1	Plotting	PyPI
pandas	Library	1.4.1	Plotting	PyPI
seaborn	Library	0.11.2	Plotting	PyPI
ijson	Library	3.1.4	Iterative JSON parsing	PyPI
tqdm	Library	4.62.3	Overseeing Progress	PyPI
Docker	Virtualization	20.10.7	Encapsulating Components	official source
PostgreSQL	DBMS	13.4	Analyzing results	Docker Hub
Ethereum Mainnet	Blockchain	multiple	Data Source	-
Mayflies	Metadata	-	Validation	Website
solc	Compiler	0.4.10-0.8.8	Evaluation	GitHub

[1] Modifications were made to these tools. See subsection 3.7.3 for details.

Table 4.2: Relevant tools and frameworks used during the course of this thesis.

	Full dataset	**10 % Sample**	**5 % Sample**	**Vulnerable Contracts**	**Standard Contracts**	**Wallet Contracts**
Size	**233 533** **skeletons**	**23 318** **skeletons**	**11 628** **skeletons**	**24** **contracts**	**412** **contracts**	**893** **skeletons**
thereof CFGs[1]	228 852	22 830	11 374	24	404	876
thereof sliceable[2]	228 693	22 813	11 368	19	396	876
extracted slices	29 280 245	2 925 905	1 452 403	1 468	1 896	120 161
represented contracts	6 500 286 [3]	56 673	13 118	-[4]	-[4]	1 203 457
avg. slices / contract	128	128	128	77	5	137

[1] There may be several reasons why EtherSolve does not extract a CFG for every contract. One reason is that some contracts do not actually store valid EVM code. A more realistic take might be that EtherSolve can simply not extract CFGs for all contracts. See https://github.com/SeUniVr/EtherSolve/issues/4 for more information.
[2] Our slicing tool may fail for some CFGs, e.g. when no slicing criterion is present.
[3] The reduction of 86 % from **47 M** contracts to **6.5 M** is explained by the ~5k skeletons for which EtherSolve was unable to create CFGs. These include skeletons for mass deployed contracts like gasTokens or mayfly contracts [dAS19b] and are representative for ~**40 M** contracts.
[4] These synthetically generated contracts do not represent any existing contracts on the real blockchain.

Table 4.3: Dimensions of our three datasets. A decline in contracts is observable after each stage.

4.3 1-to-1 Slice Matching

In order to perform *1-to-1* slice matching, we proceed the same way as [HHY$^+$21] and build a dataset of slices extracted from known-to-be vulnerable contracts. To this end, we work with contracts that previously have been classified as vulnerable by Rameder et al. [RdAS22]. Due to the difficulty of obtaining versions of solc below 0.4, we focus on contracts that were created for higher versions. This also allows us to benefit from the specified *pragmata* in order to identify a valid version of solc with little effort. We proceed by slicing the vulnerable contracts separately from the corpus, identify those slices that embody most of the vulnerability's logic and merge them into the corpus before reaching the embedding stage. Table 4.4 lists the vulnerable contracts we use in this experiment, where the first column specifies the name of the vulnerable contract, and the second column the type of vulnerability. For more information about these vulnerable contracts please refer to [RdAS22]. As an example, Figure 4.1 depicts a slice vulnerable to an *Externally Forced Fail (EFF)*[1].

After embedding, the slice vectors are compared to each other using Equation 3.1. Since this experiment is not as computationally challenging as the others, we will work with all three datasets, including the complete dataset, for which we will compare the few vulnerable slices to all 29 M slices. We select the threshold $\mathbf{t} = \mathbf{0.95}$ and extract all slice pairs whose similarity exceeds $\mathbf{t}$.

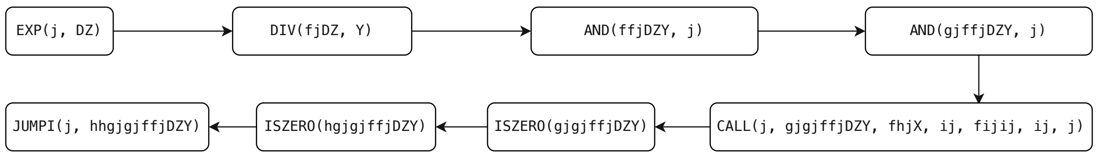


Figure 4.1: Example for a slice of a vulnerability of type EFF.

Using a relational database, we then query the best match for each of the vulnerable slices. We perform sanity checks on these matches by verifying whether the two slices do in fact resemble each other, and the instructions required for the vulnerability to exist are present. Once a pair is found whose match seems plausible, we proceed to manually inspect the source code of the matched contract, if available. For this we rely on the external service EtherScan[2] as it is a handy tool to quickly query the source code of smart contracts.

Tables 4.5, 4.6 and 4.7 display our results. We leave them uncommented for now and present analyses building upon this data in chapter 5.

[1]https://swcregistry.io/docs/SWC-113
[2]https://etherscan.io

Contract	**Vulnerability**	**#Vulnerable Slices**
QIUToken	Origin	5
UNITDummyPaymentGateway	Origin	4
UNITPaymentGatewayList	Origin	4
UNITSimplePaymentGateway	Origin	3
CrowdSalePreICO	UncheckedReturnValue	3
CreditDepositBank	UncheckedReturnValue	2
EthBird	ExternallyForcedFail	2
Masker	DelegateCall	2
EthMashChain	BlockInfoDependency	2
EthMashMount	BlockInfoDependency	2
SysEscrow	Origin	2
PrivateBank	Reentrancy	2
PrivateDeposit	Reentrancy	2
Private_Bank	Reentrancy	2
IceRockMining	ExternallyForcedFail	2
Vault	Selfdestruct	1
PiggyBank	Selfdestruct	1
TokenERC20	Selfdestruct	1
MyAdvancedToken	Selfdestruct	1
Sum		**43**

Table 4.4: Candidate vulnerable contracts. We tried to identify at least one slice per contract, but note that not all slices incorporate a vulnerability equally well.

1-to-1 similarity — 5 % sample — sanity checked			
Vulnerability	**Matched Candidate Contract**	**Score**	**Sane**
Selfdestruct	0x9f751aaacc74e55a27a19419c332e02aa96ed961	0.990336	✓
Delegatecall	0x5aeb706c39a76c31fa89bf726de1a6f7d6bc1a51	0.958257	
Delegatecall	0x8fe028eb002bbc3ec45c5df8acfff67ec95b6f88	0.991465	
ExternallyForcedFail	0x34ea8cdc7837d3a84f5869909104bdb1a7c8cb35	0.991206	✓
ExternallyForcedFail	0x0347cd66ea7756377028e494e92845c800ee1521	0.992572	✓
Origin	0xcd9e13b2f3bfc26cd99989fb849a2751932595c4	0.987938	
Origin	0xfeae69049d5a7fe9af32a0aad6e8cb77f99aac0d	0.991296	
Origin	0xe36d1b67696c8567a3997858cb73de40c9b6888f	0.991067	
Origin	0x7600beeee1db4e7e222e765f831b1afa87cc62f2	0.982846	
Origin	0xfd810ccff10dba53c619806940a4acf4416ddbe0	0.992502	
Selfdestruct	0xead66d97b7a4918163fe24bff8be5be465bac246	0.989936	
Origin	0x6d6c14d241a1c610b5d248be778c17ee57679a8f	0.990943	
Origin	0x066128b9f7557b5398db3d4ed141f2e64245ffa1	0.993294	
Selfdestruct	0x6816184f231aa6af7f959d99ec0ace5731ab33f0	0.991337	
Reentrancy	0x62195cfda73f99e1bff4881fdcfbf5c9576d3c88	0.982594	
Reentrancy	0x5bd21d52421b40affd00aabd55f46e90b2f7a32a	0.983811	✓
BlockInfoDependency	0x3edc3ca0135eca1b1b79f33ede36642f23ea5c5e	0.985866	
BlockInfoDependency	0x26b4fd727cd0ca88d4bb8150866d7c2e2d1c44d9	0.972862	
BlockInfoDependency	0x8535ef070a685eeece7dfaab5d78948065fb81cc	0.985387	
BlockInfoDependency	0x6bce536e0a938dc19dbcbb045b46b93c3d13620b	0.981298	
Reentrancy	0x2dcfce5df534b5bd27d3f1cf78e17d67addd0bce	0.984777	
Reentrancy	0x6d647bde7d25c920e92b77e9654e5654c877561d	0.983952	✓
ExternallyForcedFail	0xff93908c8e92181d623f4a58bceb5bf53fb143c5	0.982666	
Reentrancy	0xead66d97b7a4918163fe24bff8be5be465bac246	0.98828	
Reentrancy	0x900a979cfcc4a9e5f0dcac1f7cc629873e2528ec	0.979604	
UncheckedReturnValue	0x76ea2186182e3ec27c2d9c7394b83e5c8f2cf6c4	0.974858	
UncheckedReturnValue	0x5221537b92b0a405d244dec5b6d9435c1c910350	0.985035	
UncheckedReturnValue	0xbb36d6d8b2eb7266f7dc484885eed76242d0435a	0.990657	
Selfdestruct	0xc3a1d2dfee2229893cabf9d4ee0fb9d6afb963b5	0.991385	
UncheckedReturnValue	0x8e4f34b5dae571d2193578fd2615fc0200aff28c	0.962468	
Origin	0x2c84b7f585af341b38f63f18547f8cbf25316ccd	0.990265	
Origin	0x2ef624da1c1644e673b5d299ae67dc08e419ee19	0.990583	
Origin	0xe3f64dc522a66405c51d96aae234217a03502bb4	0.986649	
Origin	0x0d180978830fc88736b3664d47337c406347c147	0.984917	
Origin	0x18ecc2461dfd84c5ce9da581aca58919a8750ae5	0.987155	
Origin	0x18ecc2461dfd84c5ce9da581aca58919a8750ae5	0.986463	
Origin	0x0d180978830fc88736b3664d47337c406347c147	0.989803	
Origin	0x008fac90c491f3f01e0131241e5e9c8ccbf83a12	0.989026	
Origin	0x18ecc2461dfd84c5ce9da581aca58919a8750ae5	0.99145	
Origin	0x18ecc2461dfd84c5ce9da581aca58919a8750ae5	0.99029	
Origin	0x9d1aebde5973fc8c121103deddf0328f70b11b2e	0.99024	
			5/41

Table 4.5: Vulnerable slice matching — 5 % sample.

1-to-1 similarity — 10 % sample — sanity checked			
Vulnerability	**Matched Candidate Contract**	**Score**	**Sane**
Selfdestruct	0xedb69fe5c154eb74d1a5d48203b43086d812ad77	0.990925	
Delegatecall	0x285b2af01b71074bcebb4042fa1cee21d2f60873	0.991661	
ExternallyForcedFail	0xf547229a3b21c525630eda4fa334fada82464358	0.990225	✓
ExternallyForcedFail	0xacdb43d57fbea59d7aa4e9e6fd274ea78d0610cb	0.991428	✓
Origin	0xcd2f1aa6e9421e98459b70d6b02a710ecee3da12	0.989612	
Origin	0x96e7d5c04a5ad54ef081f16bb535443b4c017c37	0.989372	
Origin	0x078dfa9f511eaeafaec0ddb396632f4b22a2ee51	0.988888	
Origin	0x6cbae4ffd478786f1dae7359b4965cdce0fda659	0.98176	
Origin	0x9dd8db08a907ddf82eb539bb0645d1237e9024ee	0.990799	
Selfdestruct	0xbd0ea1ad69665c4108ef2e971c58b2068c0cdf97	0.983596	
Origin	0xc8d9b34e5913cd935095cae813dd345a6896c11c	0.99175	
Origin	0x1e01d81ab996fd409004e5dcb312e01fd47a83df	0.992919	
Selfdestruct	0x738dfaf60910ebcb4cd369cb983b5d36467e9673	0.98839	✓
Reentrancy	0x77a2bf0bda9775fb3524a6720dd3b16bd455e2c2	0.984812	
Reentrancy	0x95aae0975f1606dd895b58cbc1ffc4c5da5e2191	0.981802	✓
BlockInfoDependency	0xd97660f7db31024c7c48126fcb8fa35a565b8cda	0.981974	
BlockInfoDependency	0x1444073090cd558749f5289f224fd83846676e4d	0.969281	
BlockInfoDependency	0xd97660f7db31024c7c48126fcb8fa35a565b8cda	0.979826	
BlockInfoDependency	0x6ca044b9677f64382841b2996b782f7a967cb555	0.972989	
Reentrancy	0xba8d9b2bce809427b4bd8b054910108b916a8537	0.982535	
Reentrancy	0xfb30a5154b3d1cd024319d3092b3708443aee960	0.980025	
ExternallyForcedFail	0x20d14e391a80dfa8e28778c263e41e780fb8f4b8	0.978045	
Reentrancy	0x7aee02f10f41fbc8645b3e4fb505c2894a414467	0.98423	
Reentrancy	0x2ed6dac2b01a2a27803d6fe4f8e9729e92a8dfcf	0.982067	✓
UncheckedReturnValue	0x40730f34668afcb3884f050cbc3d376a444bbe44	0.977421	
UncheckedReturnValue	0x1385deab431450f67aae72ab611fad48fadb0631	0.990852	
UncheckedReturnValue	0x844224a22eed9f7787838c7ad903c596d565a573	0.98802	
Selfdestruct	0x3e4f5dd1be2db446f4ddbdb1e4b2be0e58bbb408	0.99186	
UncheckedReturnValue	0x242dd24ea38c56f345d207e2fd728723daf56fe3	0.955968	
Origin	0xad56273c8268972341aaa9e2d999e4d0a00ec0f9	0.987735	
Origin	0x3114360f81c643c2f18346a06b56d8ddd5573583	0.988753	
Origin	0xad56273c8268972341aaa9e2d999e4d0a00ec0f9	0.991081	
Origin	0x957640e9cdad57030c37b81ebdcc18988d14951f	0.982627	
Origin	0x957640e9cdad57030c37b81ebdcc18988d14951f	0.986654	
Origin	0x250057812d65e2359b980f0ee02814b641637c65	0.98884	
Origin	0xd4010881cfab9c385a33db0d7797941be834c0eb	0.987256	
Origin	0xedffbcd5ba7d78e82f55445f1dd5fc5b2d594a06	0.990752	
Origin	0xc03744f29ba7563c6d80e3f1af374866056cdca4	0.98345	
Origin	0xe041340b3338e1f220c10e9971aa4edf9bfd776e	0.985577	
Origin	0xb15657374327eb26b7fbafdc7cc765130371d49b	0.990881	
			5/40

Table 4.6: Vulnerable slice matching — 10 % sample.

1-to-1 similarity — full dataset — sanity checked			
Vulnerability	**Matched Candidate Contract**	**Score**	**Sane**
Selfdestruct	0xa5c65c70e17747371fe105c1a6fe50e4fcec246f	0.978934	✓
Delegatecall	0xb1fd7151cb2869bde7a0d148d45ca2cd6f71b7bb	0.97617	
ExternallyForcedFail	0x8b48cb5d71ae681a5fbba2064a330afbc448aaa5	0.982287	✓
ExternallyForcedFail	0x2ca103f6c1b5bdc36118c05491eea85080e93d14	0.98179	✓
Origin	0xa8e5ea57f875d8cb5700eb270238ce52ede9cad8	0.978875	
Origin	0xbca69aa1269207844783e7f9c429e05ad116b72f	0.984917	
Origin	0x40ab332dd48f35ebd227708ef381c946c4959eb6	0.980746	
Origin	0x77dfdda69e9491c7c8d78a9e2413562b32aef2ff	0.975914	
Origin	0xfe246f3ecea5765cfd3f8ac266718ea6c3165ee5	0.989504	
Selfdestruct	0x7236aef748e70f2abfb4dc21c147b3ffea07c57b	0.963273	
Origin	0xfef3fe6e8b08c8930e447e714a8eff2cc54e44cf	0.975827	
Origin	0xce75d331bed26b8d7df9818441781b9943e27a0d	0.982536	✓
Selfdestruct	0xa38a4ecb19982ad05c00f0a2419ccdd228e2e5b4	0.970559	
Reentrancy	0x44ed1b079275188a8e40e113a2474b872971ff4a	0.964864	✓
Reentrancy	0xa428680d6aec0ae8666a331d78284905601e1ca2	0.972162	✓
BlockInfoDependency	0xf629cbd94d3791c9250152bd8dfbdf380e2a3b9c	0.973201	
BlockInfoDependency	0xb7b1b594c4c9f3c692b94d28349dab7c4b261c8c	0.971125	
BlockInfoDependency	0x5d433518b080633a079381e3b3a4fb36f5458577	0.955194	
Reentrancy	0xd40d45c246bed33e2edf93b9b78119e7f993415b	0.970448	
Reentrancy	0x81e17be1afbee982250190d9acd2f14d6226bc2b	0.978076	✓
Reentrancy	0x93d21366305f42f473b7cb65fed601477965dd3f	0.965493	✓
UncheckedReturnValue	0x258c65ca3f47f13903db9dc9998294bcbe499179	0.976005	
UncheckedReturnValue	0x625c999d88891143a9e8964ce949b5ecac3d03e9	0.969992	✓
UncheckedReturnValue	0x400a13360fba517421d1b7c469c7526e2af1a4a6	0.969096	✓
Selfdestruct	0x6bf93bec191169bd09594cf5704574c9cc0dd92a	0.980715	
UncheckedReturnValue	0xf644a47ab6089eb0b4aa2b83c23a7011f5039b91	0.959201	
Origin	0xd16fa899d1a192c2cf844e956885ed09daa19dc4	0.972511	
Origin	0x682541027a628f99e8d11a6f0b93bfdfc194225b	0.982736	
Origin	0xfaf5c17a22026dc81d3b93d3c32510b23bb407ff	0.979137	
Origin	0x103c416afa628e78afb11a4c11131db1d3a5f607	0.989001	
Origin	0xb811e413920e4a4ad32d41c0ee34cb2a14873691	0.982705	
Origin	0x41c663009260aa52f0b0955cda5f4c70b2c18d70	0.981659	
Origin	0xfaf5c17a22026dc81d3b93d3c32510b23bb407ff	0.980112	
Origin	0x0c7b8eedd028f3730e7b0f3dc032e916302f09b4	0.989278	
Origin	0xbc7b80fd304d48cf4e3ea0865bf62dd12734b943	0.984084	
Origin	0xffd0444c2d34101b4acc642509c5ff59fa26d0f7	0.979814	
Origin	0x42059697e8800576eaf454fc8c2380302a85fb8a	0.97367	
			10/37

Table 4.7: Vulnerable slice matching — full dataset.

4.4 N-to-M Slice Matching

Contract Matching

The second goal of our work is to explore how well our method is suited to find similar contracts in general. To be able to make confident claims about the quality of contract matching, we use methods described in subsections 4.4.1 and 4.4.2. Since embedding frameworks usually work better when trained with larger datasets [MCCD13], we want to investigate performance differences between our experiments conducted with either of the sample sizes (**5 %** and **10 %**).

While the method for n-to-m matching is the *heuristic* method as described in section 3.5.2, we consider two different definitions of a *match* in our two experiments. The first one is a simple, asymmetric match as is part of our extraction procedure. Asymmetric in this context means that in a match $\overline{m}_{i,j}$ between two contracts i and j, the contracts are not interchangeable, i.e. $\overline{m}_{i,j} \neq \overline{m}_{j,i}$. In other words, our heuristic may determine contract i to be similar to contract j, but the inverse may not necessarily hold true. This notion of a match is used in our first experiment with *standard contracts* for the reason that contracts within this dataset are minimal implementations with little program logic. Since few slices are extracted of these contracts on average, we think comparing with real contracts only makes sense when done so in one direction.

Our second definition of a *match* tries to symmetrize the above-mentioned asymmetric matches, and is used in our second experiment. Here, we compare *wallets* which share more similarities with each other than with other contracts and also yield a high average slice count. To this end, we only regard results where both $\overline{m}_{i,j}$ and $\overline{m}_{j,i}$ exceed the threshold $\mathbf{t}$. By doing so, we "symmetrize" matches and remove those contract pairs where $\mathbf{t} > 0.9$ fails to hold true for both directions of the match. For our second experiment, we define the similarity between two values as $M_{i,j} = \min(\overline{m}_{i,j}, \overline{m}_{j,i})$, and regard such matches as symmetrical, i.e. $M_{i,j} = M_{j,i}$, while still making use of their individual matching scores $\overline{m}_{i,j}$ and $\overline{m}_{j,i}$.

4.4.1 Standard contracts

By taking reference implementations of the ERC contracts and including them in the corpus of contracts, we might be able to detect contracts of similar nature. In order to account for the variability introduced by the solc compiler on the one hand, and for the different contract versions that can be found in the wild on the other hand, we compile different combinations of both. Furthermore, we use the argument `--optimize-runs` on solc to compile each individual combination with four different levels of *lifetime gas usage* optimization: **1**, **200** (default), **10 000** and **1 000 000**. We source our selection of standard contracts from OpenZeppelin's GitHub repository[3]. Table 4.8 shows the name of the contract, which versions we were able to extract of it, and which versions of solc we used to compile them.

[3]https://github.com/OpenZeppelin/openzeppelin-contracts

The rationale behind using standard contracts for contract matching is that many smart contract developers rely on open-source reference contracts to write their own. However, an even more compelling argument is that ERC contracts must implement their respective standard interfaces to be regarded a valid ERC contract. Since function signatures are identifiable even in closed source contracts, this makes ERC contracts theoretically useful for verifying the performance of our method.

Table 4.8 displays the selection of contracts that we have extracted, and the variations of parameters we compile them with. After completing our process of compiling, generating CFGs, and slicing, we obtain 1 896 slices of 396 distinct contracts, which we proceed to match against contracts in our corpora. Since individual contracts may implement more than one interface, we define a match as *plausible* if there is a single common interface that both contracts inherit from, i.e. both of the contracts expose all function identifiers of the same Solidity interface. Tables 4.11, 4.12 and Figures 4.3a, 4.3b display this circumstance for the 10 % dataset.

We focus on two subsets of matches: One subset comprises matches between our synthetically generated standard contracts *only* (Figures 4.2a and 4.3a and Tables 4.9 and 4.11), meaning that both contracts in a match will be one of those contract variations listed in Table 4.8. The other subset comprises matches between synthetically generated contracts on one side, and real contracts on the other (Figures 4.2b and 4.3b and Tables 4.10 and 4.12). Both of these subsets are obtained during the same embedding process using the same dataset.

ERC Contract	**Contract Pragmata**	**solc Versions**
CrowdsaleMock	^0.4.24, ^0.5.0	0.4.26, 0.5.17
ECDSAMock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.0, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC165Mock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.0, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC20BurnableMock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC20MintableMock	^0.4.24, ^0.5.0	0.4.26, 0.5.0, 0.5.17
ERC20Mock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC20PausableMock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC721FullMock	^0.4.24, ^0.5.0	0.4.26, 0.5.17
ERC721MintableBurnableImpl	^0.4.24, ^0.5.0	0.4.26, 0.5.17
ERC721Mock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.8
SafeMathMock	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.0, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC20	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.0, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC721	^0.4.24, ^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.4.26, 0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.8
ERC1820Implementer	^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.5.0, 0.5.17, 0.6.0, 0.6.12, 0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC777	^0.5.0, ^0.6.0, ^0.7.0, ^0.8.0	0.5.17, 0.6.12, 0.7.0, 0.7.6, 0.8.8
ERC1155Mock	^0.7.0, ^0.8.0	0.7.0, 0.7.6, 0.8.8
ERC1155ReceiverMock	^0.7.0, ^0.8.0	0.7.0, 0.7.6, 0.8.0, 0.8.8
ERC1155	^0.7.0, ^0.8.0	0.7.0, 0.7.6, 0.8.8

Table 4.8: Commonly used contracts used as reference for this experiment. Each valid combination is compiled using four different optimization levels (1, 200, 10 000 and 1 000 000) which results in 412 individual bytecodes. Source codes for these contracts were obtained from OpenZeppelin's GitHub repository.

5 % sample ∪ standard contracts — matches between synthetic standard contracts

Contract	t=0.95		t=0.94		t=0.93		t=0.92		t=0.91		t=0.90		# slices
	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	
ERC777	270	0	307	0	317	0	347	0	378	0	380	0	14,19,20,22
ERC721FullMock	56	0	156	0	319	29	433	170	457	372	473	540	10
ERC721MintableBurnableImpl	66	0	117	0	231	10	335	69	428	228	490	383	9
ERC20PausableMock	3	0	32	0	120	0	203	10	321	50	546	123	6,9,10,12
ERC20BurnableMock	0	0	36	0	151	0	308	0	463	5	616	36	4,6,9
ERC20MintableMock	332	0	437	0	466	3	479	88	574	185	744	373	4,5
ERC1155Mock	96	6	251	50	326	168	385	341	413	446	444	538	3,6
ERC721	1 325	212	1 874	445	2 447	1 148	2 959	1 992	3 348	2 738	3 721	3 481	3,6
ERC1155	212	36	512	167	724	435	944	763	1 194	1 049	1 391	1 378	3,6
ERC20Mock	156	0	326	2	435	19	502	71	649	151	917	472	3,4,6,8
ERC1155ReceiverMock	20	23	61	51	125	99	226	139	402	212	529	322	3
ERC721Mock	964	198	1 163	329	1 300	675	1 385	1 121	1 468	1 297	1 581	1 494	2,3,6
ERC20	585	101	1 047	214	1 633	436	2 220	747	2 935	1 049	3 885	1 750	1,3,5
ERC165Mock	1 723	389	1 778	569	1 889	1 063	2 083	1 400	2 273	1 532	2 413	1 621	1,2,3
ERC1820Implementer	0	0	0	1	0	10	0	28	0	53	0	85	1
SafeMathMock	0	0	0	0	0	0	0	2	0	6	0	13	1

Table 4.9: Matches between synthetically generated standard contracts using the 5 % dataset. Figure 4.2a depicts a heatmap of relative values from this table.

5 % sample ∪ standard contracts — matches between synthetic and real contracts

Contract	t=0.95		t=0.94		t=0.93		t=0.92		t=0.91		t=0.90		# slices
	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	
ERC777	185	562	773	2 000	1 705	5 158	3 585	9 337	6 861	15 409	12 093	23 880	14,19,20,22
ERC721FullMock	762	793	1 444	2 444	2 073	5 156	2 782	8 417	3 543	12 298	4 258	17 412	10
ERC721Mintable BurnableImpl	813	740	1 409	2 210	1 993	4 657	2 665	7 530	3 359	10 668	4 026	14 812	9
ERC20PausableMock	290	131	1 049	664	2 485	1 552	5 568	3 466	10 854	7 059	18 166	11 982	6,9,10,12
ERC20BurnableMock	16	1	126	47	647	415	2 165	1 565	4 744	3 755	8 741	7 306	4,6,9
ERC20MintableMock	5 524	7 645	8 687	9 480	14 574	12 340	21 169	15 526	26 266	18 589	29 738	21 449	4,5
ERC1155Mock	46	74	422	496	910	1 140	1 576	2 271	2 366	4 796	3 227	8 176	3,6
ERC721	3 962	10 235	6 783	17 837	10 257	29 117	14 369	42 958	18 844	61 705	23 177	86 506	3,6
ERC1155	724	3 344	1 724	6 541	3 263	11 088	5 558	18 698	8 226	29 774	10 958	43 626	3,6
ERC20Mock	149	49	612	503	1 801	1 559	4 259	3 553	7 374	6 450	11 431	10 566	3,4,6,8
ERC1155ReceiverMock	556	3 146	983	5 624	1 603	8 945	2 505	14 217	3 473	20 671	4 435	28 197	3
ERC721Mock	2 247	8 296	2 940	11 021	3 737	14 089	4 794	17 349	6 105	22 623	7 584	30 533	2,3,6
ERC20	6 119	8 100	11 003	11 763	20 755	18 388	35 464	28 766	53 144	42 774	74 349	61 027	1,3,5
ERC165Mock	4 691	17 690	5 746	24 231	6 581	31 275	7 247	37 824	8 071	46 728	9 615	60 062	1,2,3
ERC1820Implementer	0	143	0	1 425	1	3 157	2	4 242	2	5 144	3	6 604	1
SafeMathMock	0	11	0	147	0	861	0	2 314	0	3 898	0	5 636	1

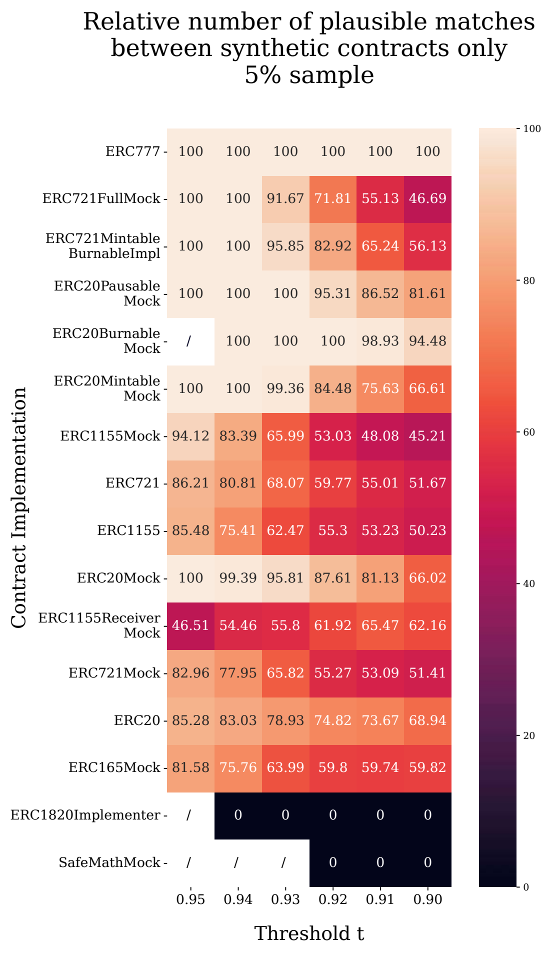
Table 4.10: Matches between synthetically generated contracts using the 5 % dataset. Figure 4.2b depicts a heatmap of relative values from this table.

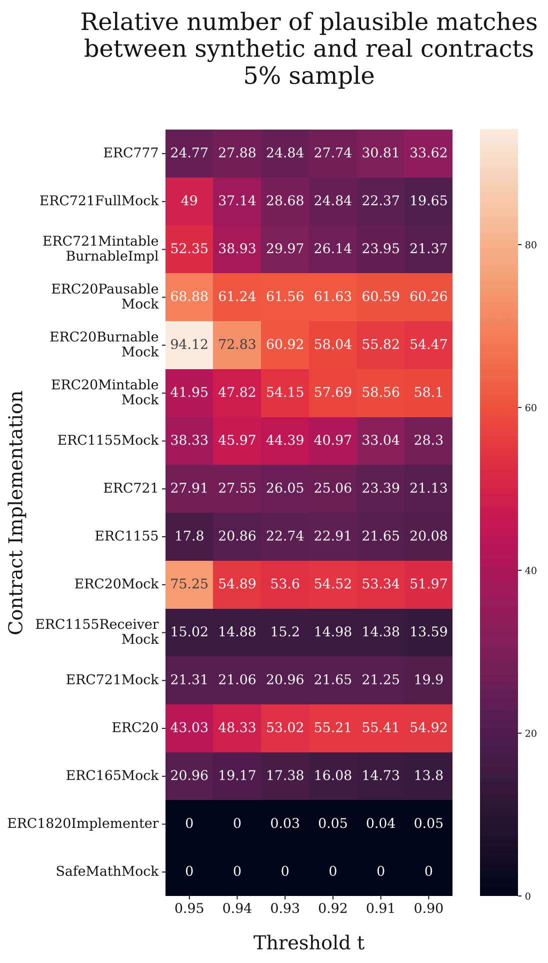
10 % sample ∪ standard contracts — matches between synthetic standard contracts													
Contract	**t=0.95**		**t=0.94**		**t=0.93**		**t=0.92**		**t=0.91**		**t=0.90**		**# slices**
	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	
ERC777	230	0	306	0	316	0	327	0	365	0	380	0	14,19,20,22
ERC721FullMock	55	0	101	0	324	8	418	87	445	207	466	393	10
ERC721MintableBurnableImpl	107	0	134	0	316	19	428	124	495	225	530	378	9
ERC20PausableMock	0	0	31	0	103	0	188	0	276	1	435	7	6,9,10,12
ERC20BurnableMock	2	0	20	0	109	0	269	0	421	2	578	21	4,6,9
ERC20MintableMock	205	0	369	0	464	0	468	45	485	139	526	224	4,5
ERC1155Mock	229	32	378	127	441	338	469	498	509	654	573	794	3,6
ERC721	235	18	415	94	629	273	826	510	990	773	1 135	1 015	3,6
ERC1155	176	17	333	89	402	230	444	422	480	560	525	697	3,6
ERC20Mock	33	0	196	0	367	1	463	8	536	43	663	125	3,4,6,8
ERC1155ReceiverMock	13	3	60	7	167	34	345	89	491	201	628	374	3
ERC721Mock	1 033	214	1 315	342	1 438	565	1 479	842	1 533	1 138	1 582	1 492	2,3,6
ERC20	83	95	266	287	552	503	784	646	967	720	1 145	786	1,3,5
ERC165Mock	1 721	403	1 777	538	1 871	711	2 109	805	2 350	1 035	2 554	1 460	1,2,3
ERC1820Implementer	0	0	0	2	0	5	0	12	0	27	10	64	1
SafeMathMock	0	0	0	0	0	3	2	14	2	32	4	72	1

Table 4.11: Matches between synthetically generated standard contracts using the 10 % dataset. Figure 4.3a depicts a heatmap of relative values from this table.

10 % sample ∪ standard contracts — matches between synthetic and real contracts													
Contract	**t=0.95**		**t=0.94**		**t=0.93**		**t=0.92**		**t=0.91**		**t=0.90**		**# slices**
	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	plausible	implausible	
ERC777	122	458	825	1 936	2 417	5 640	5 155	12 730	10 120	22 700	17 143	35 859	14,19,20,22
ERC721FullMock	739	874	2 131	2 673	3 707	7 238	5 098	13 001	6 556	18 674	7 878	24 866	10
ERC721Mintable BurnableImpl	2 148	1 910	3 595	5 935	4 950	11 416	6 273	16 929	7 612	21 974	8 744	28 813	9
ERC20PausableMock	696	257	2 205	906	4 730	1 805	9 348	3 900	16 274	8 314	26 268	15 568	6,9,10,12
ERC20BurnableMock	8	0	224	17	842	210	2 733	1 242	6 898	3 438	12 901	7 429	4,6,9
ERC20MintableMock	8 735	11 108	12 103	15 927	15 808	19 354	24 979	24 218	37 802	30 244	49 725	36 019	4,5
ERC1155Mock	78	237	894	1 056	2 481	2 465	4 458	4 662	6 572	9 489	8 104	16 795	3,6
ERC721	283	1 243	1 326	4 453	3 484	8 948	6 685	14 974	10 194	23 956	13 471	36 225	3,6
ERC1155	51	199	602	984	1 935	2 272	3 679	4 326	5 647	8 575	7 295	15 151	3,6
ERC20Mock	46	11	443	185	1 128	842	3 158	2 509	8 246	5 621	14 609	10 569	3,4,6,8
ERC1155ReceiverMock	349	1 671	1 130	5 012	2 398	10 993	4 109	19 958	6 042	31 310	8 389	44 841	3
ERC721Mock	3 976	13 428	5 463	18 079	7 448	23 586	9 925	29 174	12 760	37 190	15 607	49 562	2,3,6
ERC20	113	63	657	662	1 745	2 723	3 558	6 665	7 051	12 116	13 241	19 175	1,3,5
ERC165Mock	8 645	28 922	11 355	40 568	13 900	55 364	16 358	71 315	18 723	89 075	21 415	108 899	1,2,3
ERC1820Implementer	0	37	0	567	1	3 114	1	11 158	5	27 632	10	52 775	1
SafeMathMock	0	111	0	863	0	4 098	0	10 906	0	23 080	0	40 241	1

Table 4.12: Matches between synthetically generated contracts using the 10 % dataset. Figure 4.3b depicts a heatmap of relative values from this table.

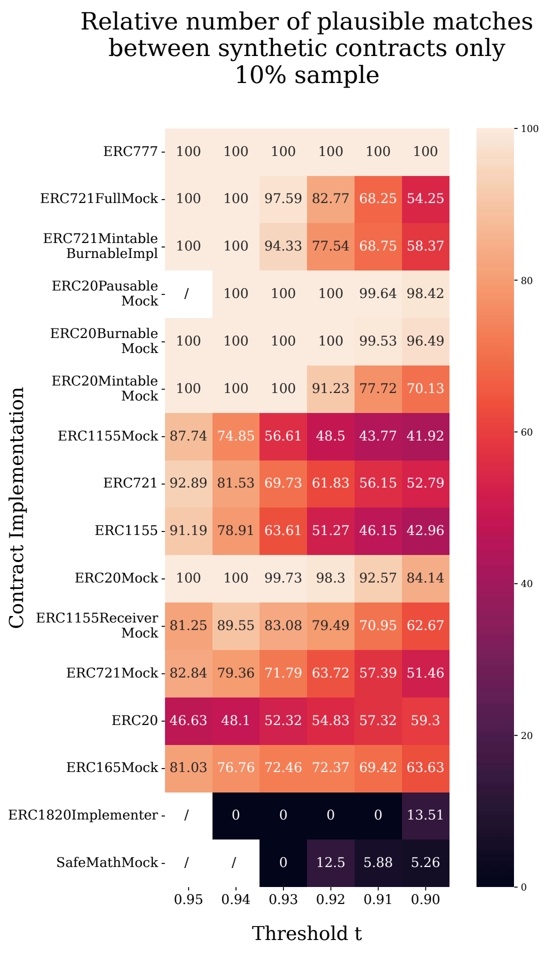




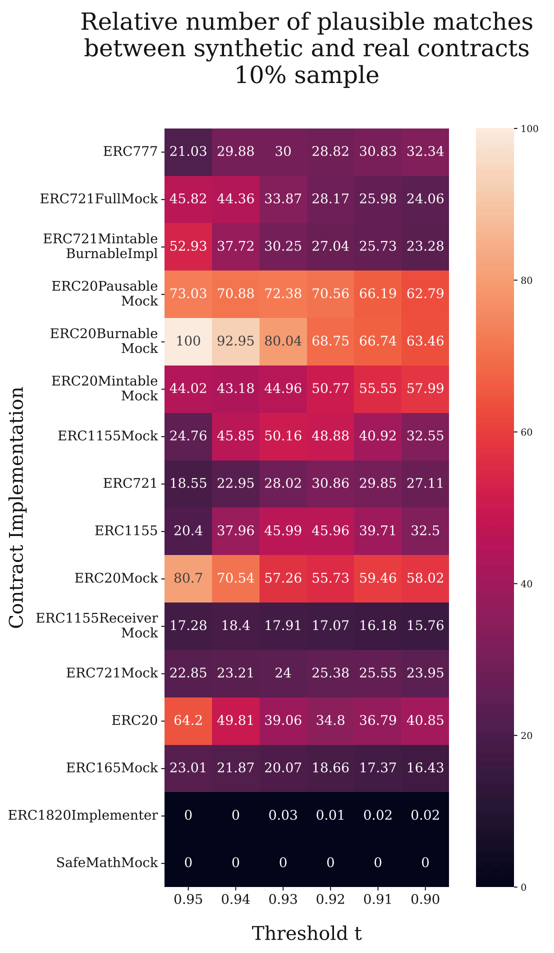
(a) Heatmap for matches between synthetic contracts only.

(b) Heatmap for matches between synthetic and real contracts.

Figure 4.2: Two heatmaps displaying the number of plausible match relative to all matches per threshold and contract. Both heatmaps represent distinct subsets of matches within the 5 % dataset.



(a) Heatmap for matches between synthetic contracts only.



(b) Heatmap for matches between synthetic and real contracts.

Figure 4.3: Two heatmaps displaying the number of plausible match relative to all matches per threshold and contract. Both heatmaps represent distinct subsets of matches within the 10 % dataset.

4.4.2 Wallet Contracts

We make use of wallets that have previously been classified as similar by [dAS20d]. The authors combine three different methods for identifying wallets. To be precise, they identify wallets based on their contract interfaces, by locating their factories, and they use the external blockchain service EtherScan[4] to extract contracts that contain the term 'wallet' in their name. This way, they distill 40 distinct families of wallets that can be further subdivided into 893 distinct skeleton groups distinguished by their deployment bytecode. We extract candidate contracts for each of these 893 skeletons, extract their CFGs and slice them before including them into the contract corpora. As was the case with our previous datasets, EtherSolve fails to extract CFGs for some contracts, reducing 893 wallet skeletons to 876. We proceed with these, check whether the wallets match with each other and whether a match occurs within a specific wallet group or not. If mismatches occur, we try to explain them by looking at source code if available. Table 4.13 displays the wallet types and the number of skeletons that were classified.

In order to be able to compare matches among groups we compute the number of pairwise matches $M_{i,j}$ where $i > j$. This eliminates symmetric matches ($M_{i,j} = M_{j,i}$) and identities ($M_{i,i}$). Equation 4.1 and Figure 4.4 provide formal and visual descriptions for this idea respectively.

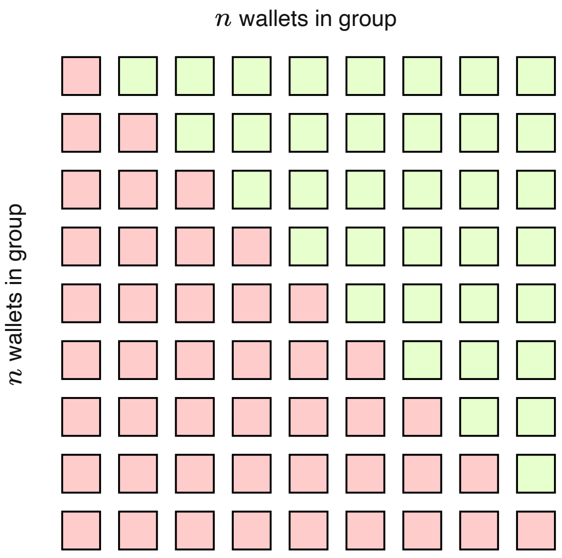


Figure 4.4: Visual depiction (and optical illusion) of Equation 4.1.

$$\text{t-sum}(n) = n^2 - \boxed{\sum_{k=1}^{n} k} = \boxed{\frac{n(n-1)}{2}} \tag{4.1}$$

[4]https://etherscan.io/

Wallet Group	**Classified Skeletons**	**Sliced Skeletons=n**	**t-sum(n)**
multisig Stefan George	345	343	58 653
multisig Gavin Wood/Ethereum/Parity	119	116	6 670
multisig WalletSimple/BitGo	98	94	4 371
multisig WalletSimple/BitGo forwarder	69	67	2 211
multisig Christian Lundkvist	44	44	946
controlled	33	32	496
smart GnosisSafe	28	28	378
timelocked wallet	16	16	120
smart Julien Niset/Argent	14	14	91
simple wallet 2	13	13	78
wallet 3	13	13	78
loopring wallet	10	9	36
consumer wallet	9	9	36
dapper	8	7	21
multisig Teambrella Wallet	7	7	21
wallet1	7	6	15
ambi wallet	6	4	10
multisig Julien Niset/Argent	5	5	10
smartwallet	5	5	10
spendable wallet	5	5	6
intermediatewallet	4	4	6
multisig Unchained Capital	3	3	3
logicproxywallet	3	3	3
multisig NiftyWallet	3	3	3
wallet 2	3	3	3
wallet 4	3	3	3
simple wallet	2	2	1
basicwallet	2	2	1
eidoo wallet	2	2	1
simple wallet 3	2	2	1
multisig Ivt	2	2	1
wallet 7	2	2	1
autowallet	1	1	0
ether wallet 1	1	1	0
ether wallet 2	1	1	0
poloniex2	1	1	0
simple wallet 4	1	1	0
ICT lock	1	1	0
wallet 5	1	1	0
wallet 6	1	1	0

Table 4.13: Wallet groups and their respective amount of classified and sliced contracts. As was the case in our previous experiments, not all of our base data was processable, hence the third column.

Number of (mis)matches per t and per dataset						
	Wallets trained using 5 % sample			**Wallets trained using 10 % sample**		
t	**matches**[1]	**mismatches**[2]	**% matches**	**matches**[1]	**mismatches**[2]	**% matches**
0.95	2 533	0	100 %	2 223	0	100 %
0.94	7 967	4	99.94 %	7 097	5	99.93 %
0.93	15 877	30	99.81 %	14 273	25	99.83 %
0.92	21 777	269	98.78 %	20 481	153	99.29 %
0.91	26 264	2 429	91.53 %	24 717	1 414	94.59 %
0.90	31 146	9 603	76.43 %	29 107	6 002	82.90 %

[1] Number of matches where both wallets were of the same type.
[2] Number of matches where the two wallets were of different type.

Table 4.14: The threshold t and its effect on matches and mismatches between wallets.

Table 4.14 shows the ratio of matches to mismatches in their respective datasets, keyed by different brackets of **t**. Symmetrical matches, i.e. $M_{i,j} = M_{j,i}$ are counted only once, and identities, i.e. $M_{i,i}$ are excluded from the statistics. Tables 4.15 and 4.16 show how our method performs on the wallet datasets. The confusion matrices in Figures 4.5, 4.6, 4.7, 4.8, 4.9 and 4.10 show which wallet groups were more prone to mismatches than others.

Optimal vs. actual vs. mismatches — Wallets trained with 5 % sample — t = 0.90

Wallet Group	**n**	**t-sum(n)**	**matches**	**mismatches**	**% matches**	**diff**	**% diff**
multisig Stefan George	343	58 653	26 567	4 352	85.92 %	−32 086	45.30 %
multisig Gavin Wood/Ethereum/Parity	116	6 670	2 445	2 700	47.52 %	−4 225	36.66 %
multisig WalletSimple/BitGo	94	4 371	985	597	62.26 %	−3 386	22.53 %
multisig WalletSimple/BitGo forwarder	67	2 211	643	378	62.98 %	−1 568	29.08 %
multisig Christian Lundkvist	44	946	109	64	63.01 %	−837	11.52 %
controlled	32	496	54	70	43.55 %	−442	10.89 %
smart GnosisSafe	28	378	112	602	15.69 %	−266	29.63 %
timelocked wallet	16	120	25	34	42.37 %	−95	20.83 %
smart Julien Niset/Argent	14	91	23	123	15.75 %	−68	25.27 %
simple wallet 2	13	78	19	2	90.48 %	−59	24.36 %
wallet 3	13	78	78	65	54.55 %	0	100.00 %
consumer wallet	9	36	30	322	8.52 %	−6	83.33 %
loopring wallet	9	36	14	9	60.87 %	−22	38.89 %
dapper	7	21	9	2	81.82 %	−12	42.86 %
multisig Teambrella Wallet	7	21	11	15	42.31 %	−10	52.38 %
wallet1	6	15	1	23	4.17 %	−14	6.67 %
spendable wallet	5	10	4	33	10.81 %	−6	40.00 %
multisig Julien Niset/Argent	5	10	1	35	2.78 %	−9	10.00 %
smartwallet	5	10	4	1	80.00 %	−6	40.00 %
ambi wallet	4	6	0	0	0 %	−6	0 %
intermediatewallet	4	6	1	27	3.57 %	−5	16.67 %
multisig Unchained Capital	3	3	1	0	0 %	−2	33.33 %
logicproxywallet	3	3	0	0	0 %	−3	0 %
multisig NiftyWallet	3	3	3	35	7.89 %	0	100.00 %
wallet 2	3	3	3	0	0 %	0	100.00 %
wallet 4	3	3	1	8	11.11 %	−2	33.33 %
simple wallet 3	2	1	1	18	5.26 %	0	100.00 %
multisig Ivt	2	1	1	19	5.00 %	0	100.00 %
eidoo wallet	2	1	1	0	0 %	0	100.00 %
basicwallet	2	1	0	13	0 %	−1	0 %
wallet 7	2	1	0	8	0 %	−1	0 %
simple wallet	2	1	0	0	0 %	−1	0 %
poloniex2	1	0	0	0	0 %	0	0 %
wallet 5	1	0	0	2	0 %	0	0 %
wallet 6	1	0	0	0	0 %	0	0 %
simple wallet 4	1	0	0	0	0 %	0	0 %
autowallet	1	0	0	3	0 %	0	0 %
ICT lock	1	0	0	27	0 %	0	0 %
ether wallet 2	1	0	0	14	0 %	0	0 %
ether wallet 1	1	0	0	2	0 %	0	0 %
	876	**74 284**	**31 146**	**9 603**	**76,43 %**	**−43 126**	**41.93 %**

Table 4.15: Wallet groups and the optimal number of matches within each group vs. the number of contract pairs that matched using our method. The table shows data using our 5 % dataset.

Optimal vs. actual vs. mismatches — Wallets trained with 10 % sample — t = 0.90

Wallet Group	**n**	**t-sum(n)**	**matches**	**mismatches**	**% matches**	**diff**	**% diff**
multisig Stefan George	343	58 653	24 955	2 641	90.43 %	−33 698	42.55 %
multisig Gavin Wood/Ethereum/Parity	116	6 670	2 157	1 325	61.95 %	−4 513	32.34 %
multisig WalletSimple/BitGo	94	4 371	931	333	73.66 %	−3 440	21.30 %
multisig WalletSimple/BitGo forwarder	67	2 211	562	279	66.83 %	−1 649	25.42 %
multisig Christian Lundkvist	44	946	118	41	74.21 %	−828	12.47 %
controlled	32	496	43	55	43.88 %	−453	8.67 %
smart GnosisSafe	28	378	111	659	14.42 %	−267	29.37 %
timelocked wallet	16	120	29	39	42.65 %	−91	24.17 %
smart Julien Niset/Argent	14	91	17	62	21.52 %	−74	18.68 %
simple wallet 2	13	78	23	9	71.88 %	−55	29.49 %
wallet 3	13	78	78	38	67.24 %	0	100.00 %
consumer wallet	9	36	30	256	10.49 %	−6	83.33 %
loopring wallet	9	36	13	5	72.22 %	−23	36.11 %
dapper	7	21	9	1	90.00 %	−12	42.86 %
multisig Teambrella Wallet	7	21	10	5	66.67 %	−11	47.62 %
wallet1	6	15	2	27	6.90 %	−13	13.33 %
spendable wallet	5	10	4	40	9.09 %	−6	40.00 %
multisig Julien Niset/Argent	5	10	1	16	5.88 %	−9	10.00 %
smartwallet	5	10	2	0	0	−8	20.00 %
ambi wallet	4	6	0	0	0	−6	0
intermediatewallet	4	6	1	25	3.85 %	−5	16.67 %
multisig Unchained Capital	3	3	1	1	50.00 %	−2	33.33 %
logicproxywallet	3	3	0	0	0	−3	0
multisig NiftyWallet	3	3	3	28	9.68 %	0	100.00 %
wallet 2	3	3	3	0	0	0	100.00 %
wallet 4	3	3	1	6	14.29 %	−2	33.33 %
simple wallet 3	2	1	1	12	7.69 %	0	100.00 %
multisig Ivt	2	1	1	20	4.76 %	0	100.00 %
eidoo wallet	2	1	1	0	0	0	100.00 %
basicwallet	2	1	0	11	0	−1	0
wallet 7	2	1	0	6	0	−1	0
simple wallet	2	1	0	0	0	−1	0
poloniex2	1	0	0	12	0	0	0
wallet 5	1	0	0	0	0	0	0
wallet 6	1	0	0	0	0	0	0
simple wallet 4	1	0	0	0	0	0	0
autowallet	1	0	0	10	0	0	0
ICT lock	1	0	0	28	0	0	0
ether wallet 2	1	0	0	10	0	0	0
ether wallet 1	1	0	0	2	0	0	0
	876	**74 284**	**29 107**	**6 002**	**82.90 %**	**−45 165**	**39.18 %**

Table 4.16: Wallet groups and the optimal number of matches within each group vs. the number of contract pairs that matched using our method. The table shows data using our 10 % dataset.

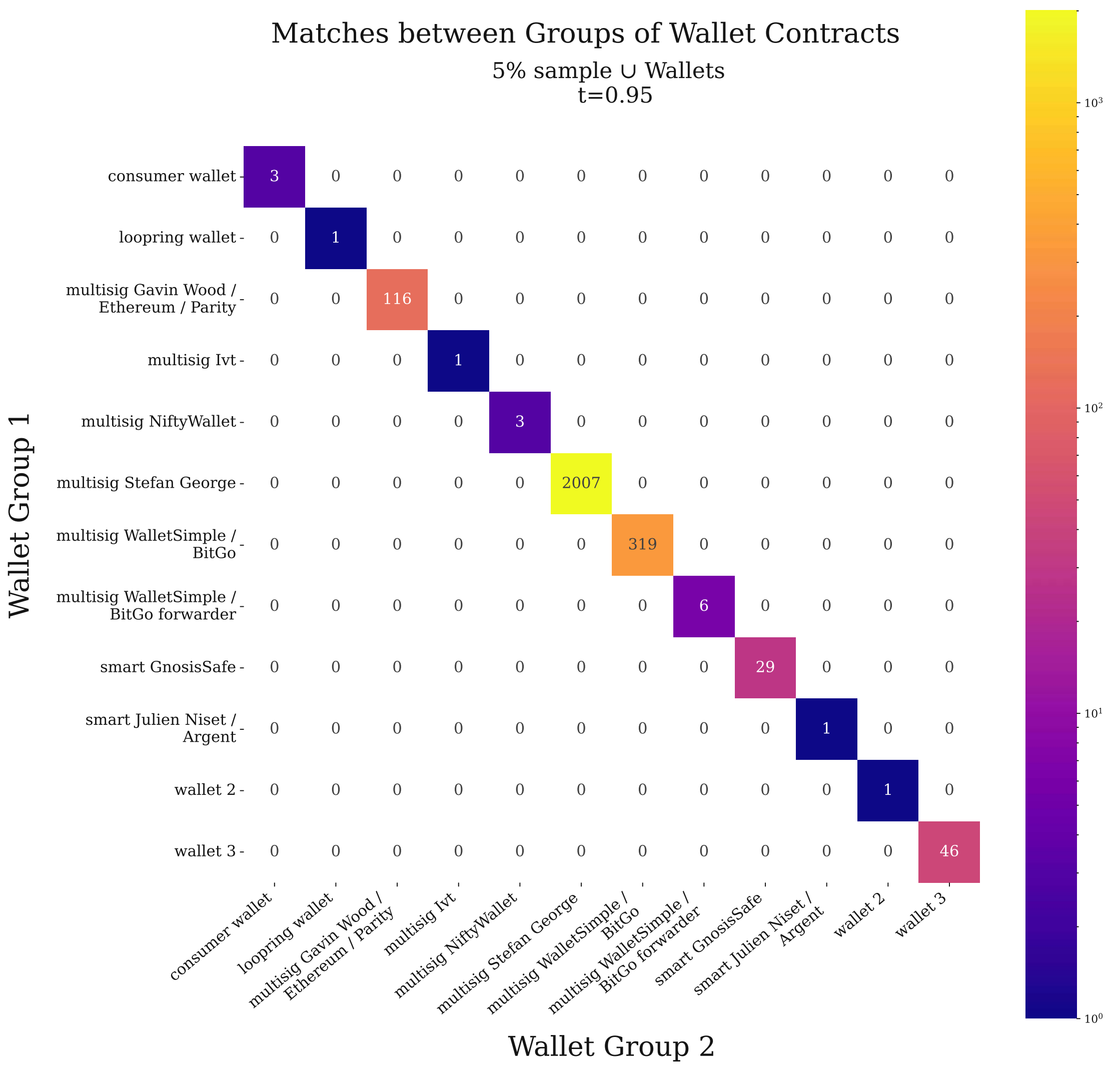
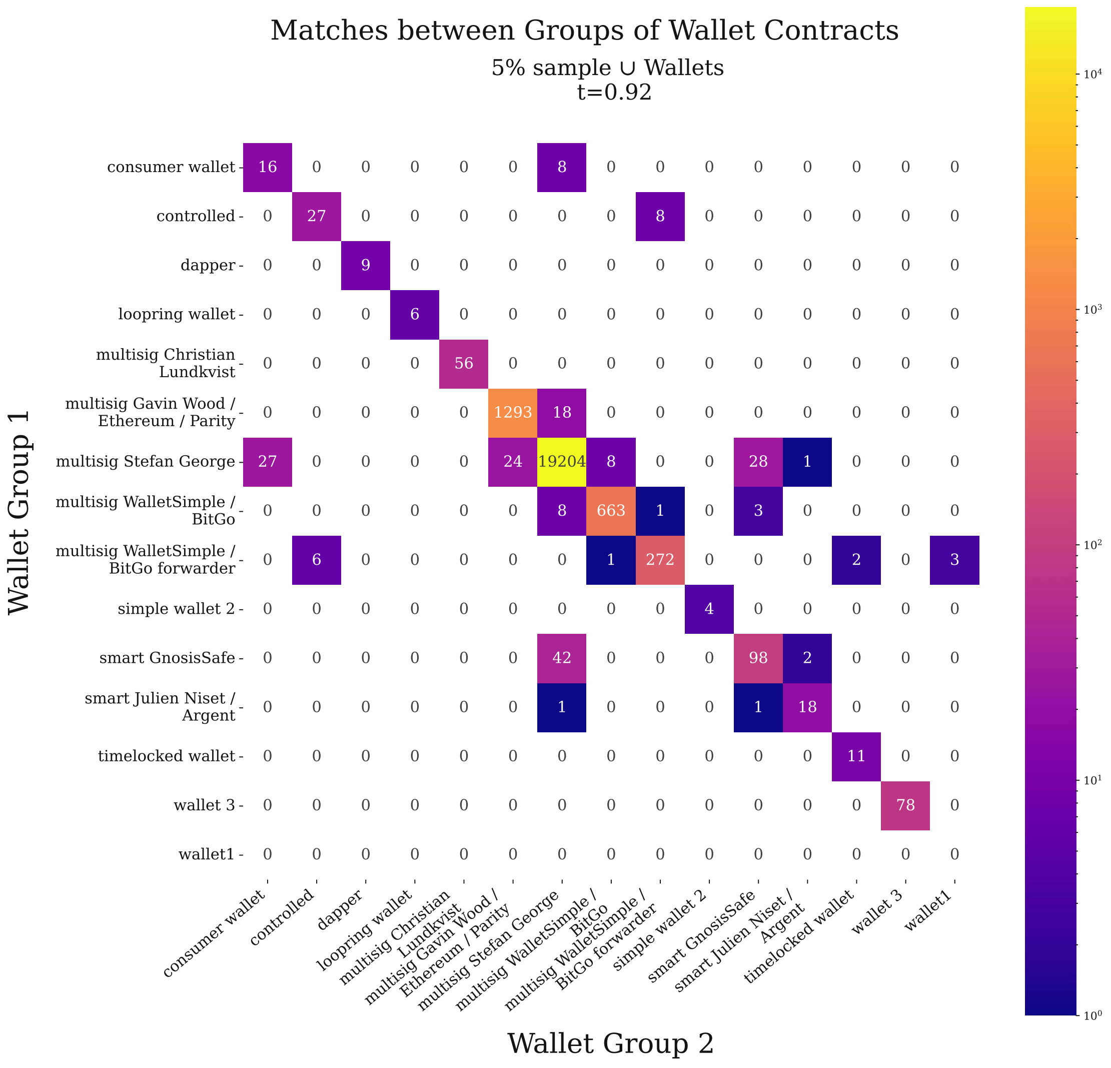
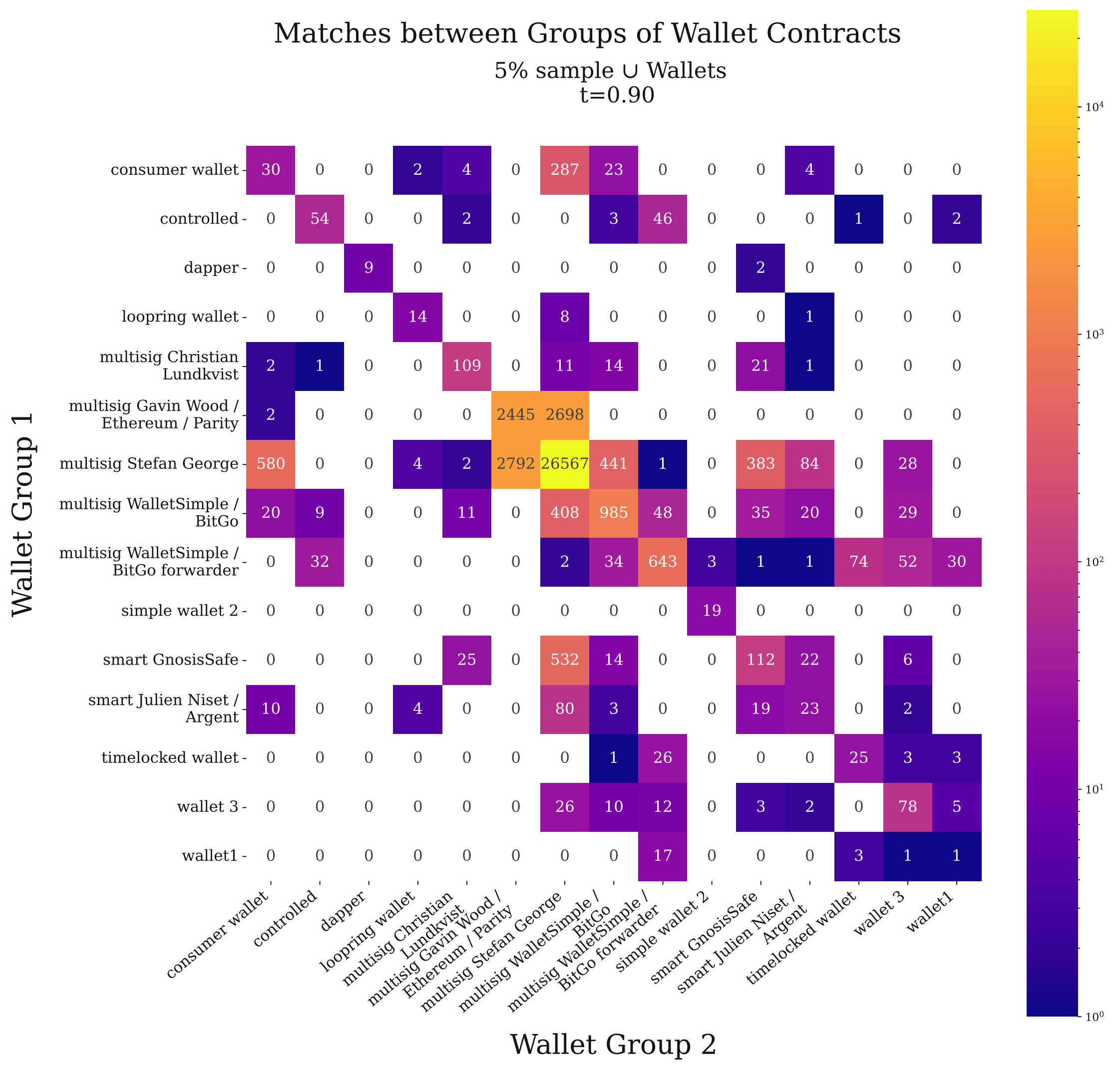


Figure 4.5: Confusion matrix of matches between wallets groups using the 5 % sample and $t = 0.95$.



	consumer wallet	controlled	dapper	loopring wallet	multisig Christian Lundkvist	multisig Gavin Wood / Ethereum / Parity	multisig Stefan George	multisig WalletSimple / BitGo	multisig WalletSimple / BitGo forwarder	simple wallet 2	smart GnosisSafe	smart Julien Niset / Argent	timelocked wallet	wallet 3	wallet1
consumer wallet	16	0	0	0	0	0	8	0	0	0	0	0	0	0	0
controlled	0	27	0	0	0	0	0	0	8	0	0	0	0	0	0
dapper	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0
loopring wallet	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0
multisig Christian Lundkvist	0	0	0	0	56	0	0	0	0	0	0	0	0	0	0
multisig Gavin Wood / Ethereum / Parity	0	0	0	0	0	1293	18	0	0	0	0	0	0	0	0
multisig Stefan George	27	0	0	0	0	24	19204	8	0	0	28	1	0	0	0
multisig WalletSimple / BitGo	0	0	0	0	0	0	8	663	1	0	3	0	0	0	0
multisig WalletSimple / BitGo forwarder	0	6	0	0	0	0	0	1	272	0	0	0	2	0	3
simple wallet 2	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0
smart GnosisSafe	0	0	0	0	0	0	42	0	0	0	98	2	0	0	0
smart Julien Niset / Argent	0	0	0	0	0	0	1	0	0	0	1	18	0	0	0
timelocked wallet	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0
wallet 3	0	0	0	0	0	0	0	0	0	0	0	0	0	78	0
wallet1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.6: Confusion matrix of matches between wallets groups using the 5 % sample and $t = 0.92$. We include the top 15 most common wallet types.



Wallet Group 1 \ Wallet Group 2	consumer wallet	controlled	dapper	loopring wallet	multisig Christian Lundkvist	multisig Gavin Wood / Ethereum / Parity	multisig Stefan George	multisig WalletSimple / BitGo	multisig WalletSimple / BitGo forwarder	simple wallet 2	smart GnosisSafe	smart Julien Niset / Argent	timelocked wallet	wallet 3	wallet1
consumer wallet	30	0	0	2	4	0	287	23	0	0	0	4	0	0	0
controlled	0	54	0	0	2	0	0	3	46	0	0	0	1	0	2
dapper	0	0	9	0	0	0	0	0	0	0	2	0	0	0	0
loopring wallet	0	0	0	14	0	0	8	0	0	0	0	1	0	0	0
multisig Christian Lundkvist	2	1	0	0	109	0	11	14	0	0	21	1	0	0	0
multisig Gavin Wood / Ethereum / Parity	2	0	0	0	0	2445	2698	0	0	0	0	0	0	0	0
multisig Stefan George	580	0	0	4	2	2792	26567	441	1	0	383	84	0	28	0
multisig WalletSimple / BitGo	20	9	0	0	11	0	408	985	48	0	35	20	0	29	0
multisig WalletSimple / BitGo forwarder	0	32	0	0	0	0	2	34	643	3	1	1	74	52	30
simple wallet 2	0	0	0	0	0	0	0	0	0	19	0	0	0	0	0
smart GnosisSafe	0	0	0	0	25	0	532	14	0	0	112	22	0	6	0
smart Julien Niset / Argent	10	0	0	4	0	0	80	3	0	0	19	23	0	2	0
timelocked wallet	0	0	0	0	0	0	0	1	26	0	0	0	25	3	3
wallet 3	0	0	0	0	0	0	26	10	12	0	3	2	0	78	5
wallet1	0	0	0	0	0	0	0	0	17	0	0	0	3	1	1

Figure 4.7: Confusion matrix of matches between wallets groups using the 5 % sample and $t = 0.90$. We include the top 15 most common wallet types.

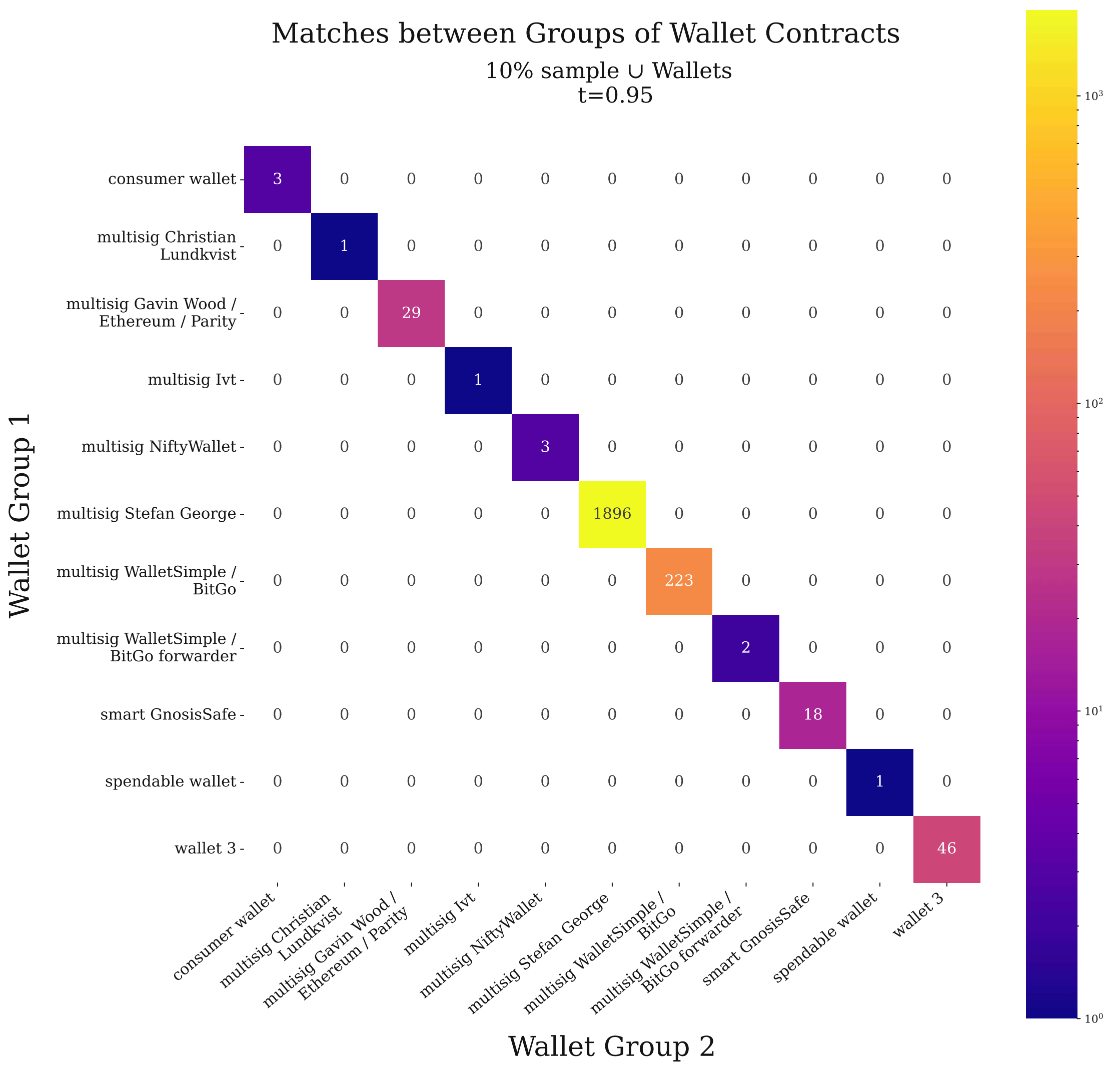
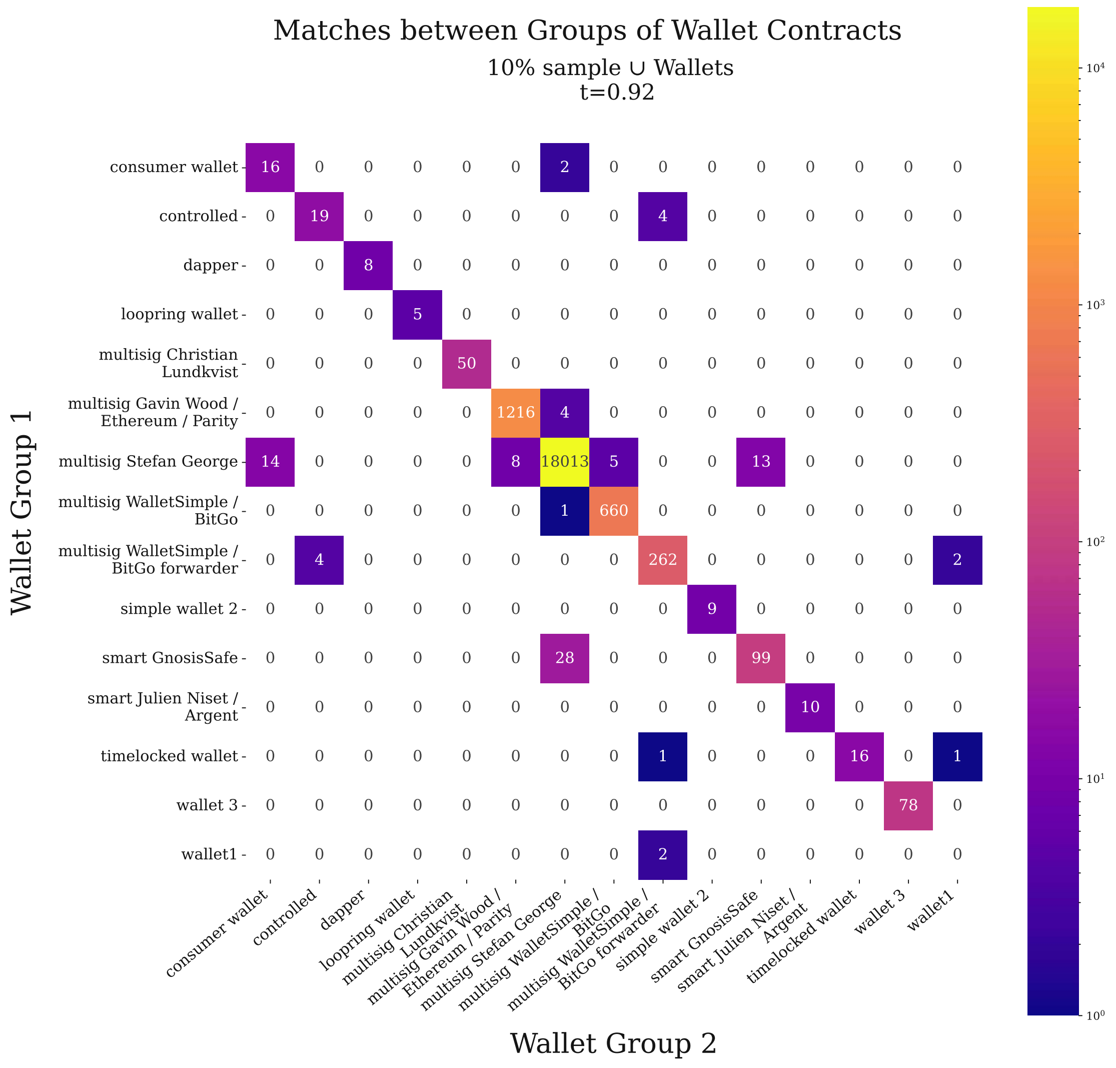
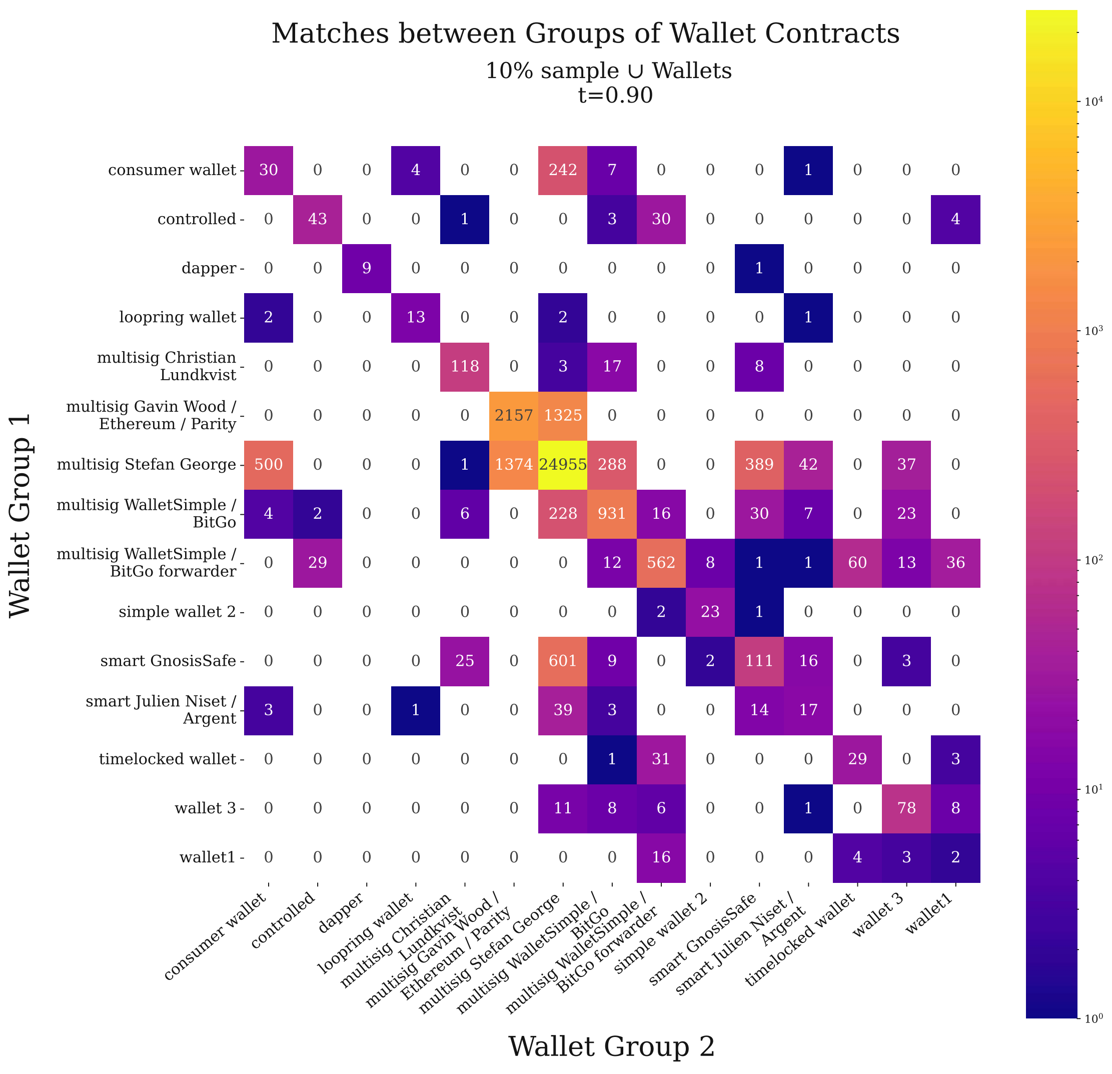


Figure 4.8: Confusion matrix of matches between wallets groups using the 10 % sample and $t = 0.95$.



Wallet Group 1 \ Wallet Group 2	consumer wallet	controlled	dapper	loopring wallet	multisig Christian Lundkvist	multisig Gavin Wood / Ethereum / Parity	multisig Stefan George	multisig WalletSimple / BitGo	multisig WalletSimple / BitGo forwarder	simple wallet 2	smart GnosisSafe	smart Julien Niset / Argent	timelocked wallet	wallet 3	wallet1
consumer wallet	16	0	0	0	0	0	2	0	0	0	0	0	0	0	0
controlled	0	19	0	0	0	0	0	0	4	0	0	0	0	0	0
dapper	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
loopring wallet	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0
multisig Christian Lundkvist	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0
multisig Gavin Wood / Ethereum / Parity	0	0	0	0	0	1216	4	0	0	0	0	0	0	0	0
multisig Stefan George	14	0	0	0	0	8	18013	5	0	0	13	0	0	0	0
multisig WalletSimple / BitGo	0	0	0	0	0	0	1	660	0	0	0	0	0	0	0
multisig WalletSimple / BitGo forwarder	0	4	0	0	0	0	0	0	262	0	0	0	0	0	2
simple wallet 2	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0
smart GnosisSafe	0	0	0	0	0	0	28	0	0	0	99	0	0	0	0
smart Julien Niset / Argent	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0
timelocked wallet	0	0	0	0	0	0	0	0	1	0	0	0	16	0	1
wallet 3	0	0	0	0	0	0	0	0	0	0	0	0	0	78	0
wallet1	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0

Figure 4.9: Confusion matrix of matches between wallets groups using the 10 % sample and $t = 0.92$. We include the top 15 most common wallet types.



	consumer wallet	controlled	dapper	loopring wallet	multisig Christian Lundkvist	multisig Gavin Wood / Ethereum / Parity	multisig Stefan George	multisig WalletSimple / BitGo	multisig WalletSimple / BitGo forwarder	simple wallet 2	smart GnosisSafe	smart Julien Niset / Argent	timelocked wallet	wallet 3	wallet1
consumer wallet	30	0	0	4	0	0	242	7	0	0	0	1	0	0	0
controlled	0	43	0	0	1	0	0	3	30	0	0	0	0	0	4
dapper	0	0	9	0	0	0	0	0	0	0	1	0	0	0	0
loopring wallet	2	0	0	13	0	0	2	0	0	0	0	1	0	0	0
multisig Christian Lundkvist	0	0	0	0	118	0	3	17	0	0	8	0	0	0	0
multisig Gavin Wood / Ethereum / Parity	0	0	0	0	0	2157	1325	0	0	0	0	0	0	0	0
multisig Stefan George	500	0	0	0	1	1374	24955	288	0	0	389	42	0	37	0
multisig WalletSimple / BitGo	4	2	0	0	6	0	228	931	16	0	30	7	0	23	0
multisig WalletSimple / BitGo forwarder	0	29	0	0	0	0	0	12	562	8	1	1	60	13	36
simple wallet 2	0	0	0	0	0	0	0	0	2	23	1	0	0	0	0
smart GnosisSafe	0	0	0	0	25	0	601	9	0	2	111	16	0	3	0
smart Julien Niset / Argent	3	0	0	1	0	0	39	3	0	0	14	17	0	0	0
timelocked wallet	0	0	0	0	0	0	0	1	31	0	0	0	29	0	3
wallet 3	0	0	0	0	0	0	11	8	6	0	0	1	0	78	8
wallet1	0	0	0	0	0	0	0	0	16	0	0	0	4	3	2

Figure 4.10: Confusion matrix of matches between wallets groups using the 10 % sample and $t = 0.90$. We include the top 15 most common wallet types.

4.4.3 Correlation Metrics

We proceed by providing various correlation metrics between matching values and the *Jaccard index* over function signatures. For this we regard results from different datasets, as described in the captions of the figures. We use the following notations:

N	:	population size, i.e. number of total matches in the respective experiment
n	:	size of sample for plotting
r	:	Pearson's correlation coefficient (over the entire population)
p	:	p-value for Pearson's correlation coefficient
$\mathbf{cr_i}$	:	creator of contract i

As the calculation of p-value assumes that each dimension is normally distributed [Kow72, p. 1-12] we proceed to conduct our correlation analyses using the entire (finite) population N, which allows us to omit significance testing for our data points. In order to achieve readable graphs, however, we limit our observations for plotting to a sample size of $n = 1000$. In other words, the specified r-values in the plots refer to the population size N instead of the sample size n.

We further want to investigate whether two contracts that are deployed by the same creator are more likely to be similar to each other than two contracts from different creators. The η^2 coefficient is used to determine correlations between nominal values, e.g. a boolean value that determines whether two contracts in a match were deployed by the same creator, and metric values, i.e. our similarity score $M_{i,j}$. Cohen et al. [Coh88, p. 282] define it as:

$$\eta^2 = 1 - \frac{E_p}{E_t} \tag{4.2}$$

where

$$E_t = \sum_{i=1}^{n} (y_i - \bar{y})^2$$
$$E_p = \sum_k \sum_{i=1}^{n} (y_i - \bar{y}_k)^2 \delta_{ik}$$
$$\delta_{ik} = \begin{cases} 1, & \text{if } i = k \\ 0 & \text{else} \end{cases}$$

In the above equation, $\bar{y}_k$ denotes the group mean of group k. In our case, we have two groups, *true* and *false*, indicating whether two contracts were deployed by the same creator or not.

Further, Cohen et al. [Coh88, p. 282-287] define as rule of thumb:

- $\eta^2 < 0.01$: negligible effect
- $0.01 \leq \eta^2 < 0.06$: small effect size
- $0.06 \leq \eta^2 < 0.14$: medium effect size
- $0.14 \leq \eta^2$: large effect size

η^2 correlation coefficient				
	5 % sample		**10 % sample**	
t	η^2	N	η^2	N
0.90	0.0020	2 698 247	0.0026	9 428 544
0.91	0.0032	1 312 887	0.0043	4 649 524
0.92	0.0053	555 119	0.0076	2 000 222
0.93	0.0097	202 166	0.0136	721 609
0.94	0.0188	59 502	0.0284	201 170
0.95	0.0321	13 334	0.0577	38 835

Table 4.17: η^2 correlation coefficient similarity score $M_{i,j}$ and creators $cr_i = cr_j$ per dataset. By increasing the threshold t, and thus regarding “higher quality” matches, the correlation becomes stronger.

Table 4.17 lists values of η^2 for different thresholds. The remainder of this section introduces calculations which we later use to discuss the hypotheses made in section 3.6.

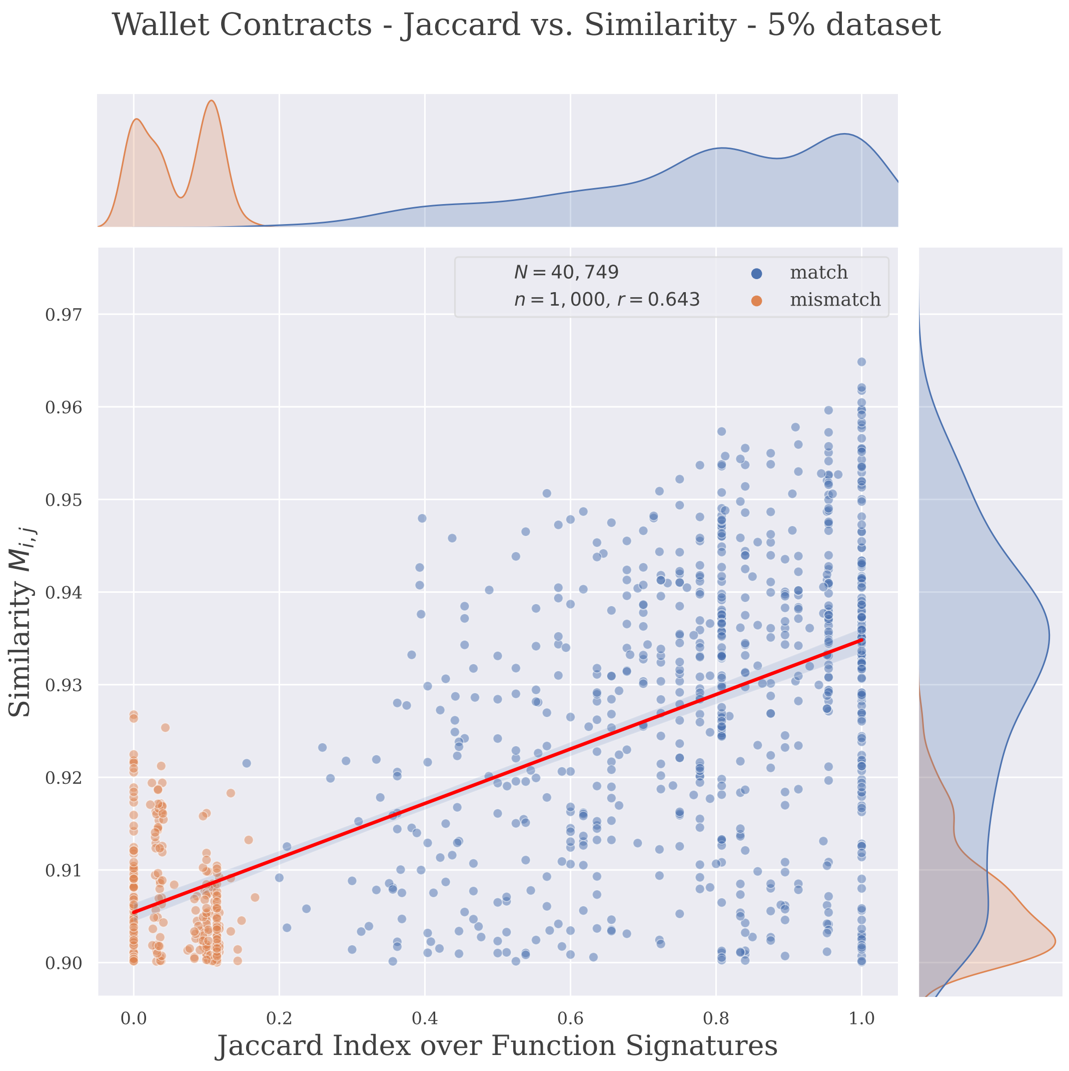


Figure 4.11: Scatter plot for matches between wallets trained with the 5 % dataset. Individual dots represent a match, and the red line represents the *line of best fit* over the sampled observations. The horizontal axis represents the Jaccard index over their function signatures, and the vertical axis the similarity score $M_{i,j}$. This makes the vertical axis synonymous with the threshold t in Table 4.14.

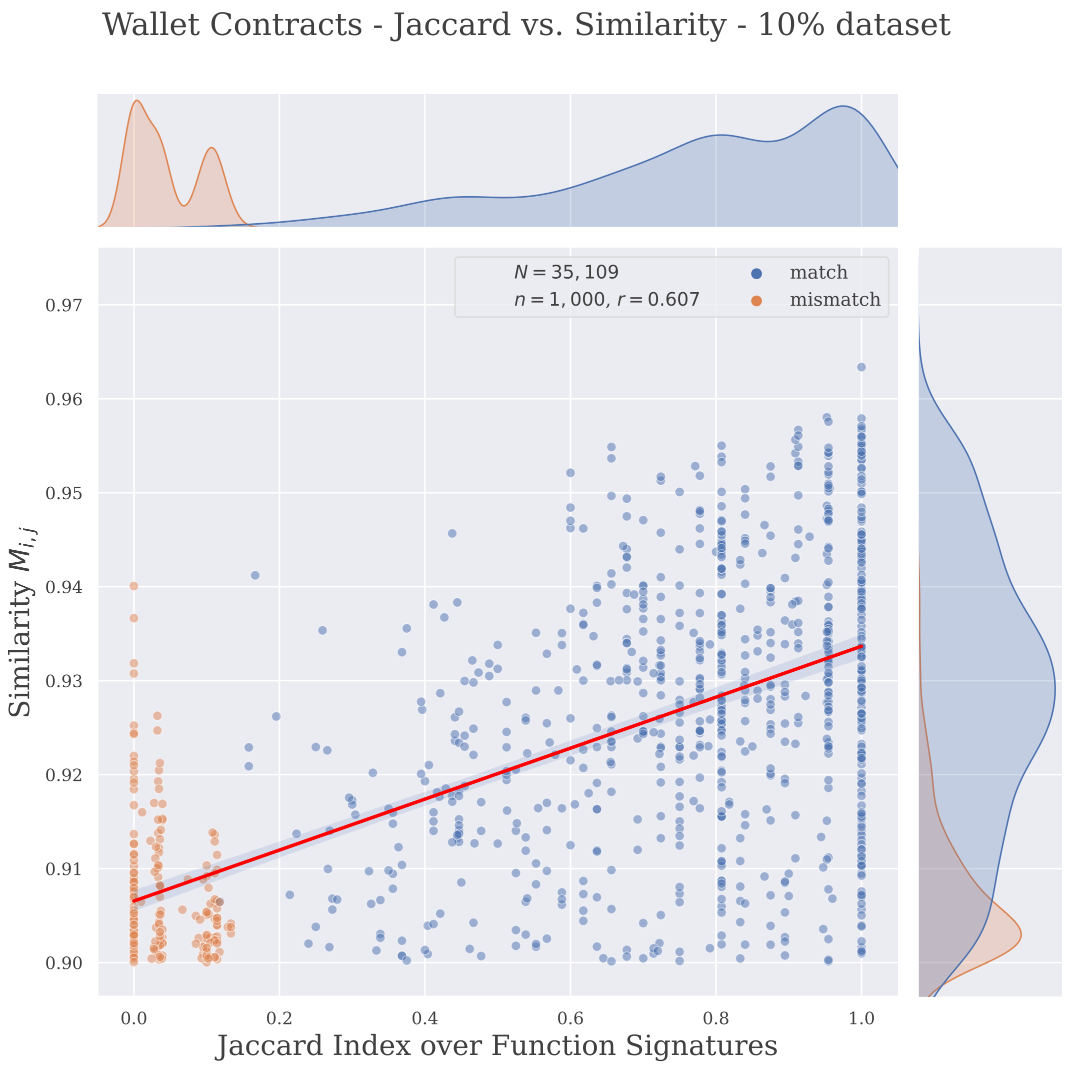


Figure 4.12: Scatter plot for matches between wallets trained with the 10 % dataset. Individual dots represent a match, and the red line represents the *line of best fit* over the sampled observations. The horizontal axis represents the Jaccard index over their function signatures, and the vertical axis the similarity score $M_{i,j}$. This makes the vertical axis synonymous with the threshold t in Table 4.14.

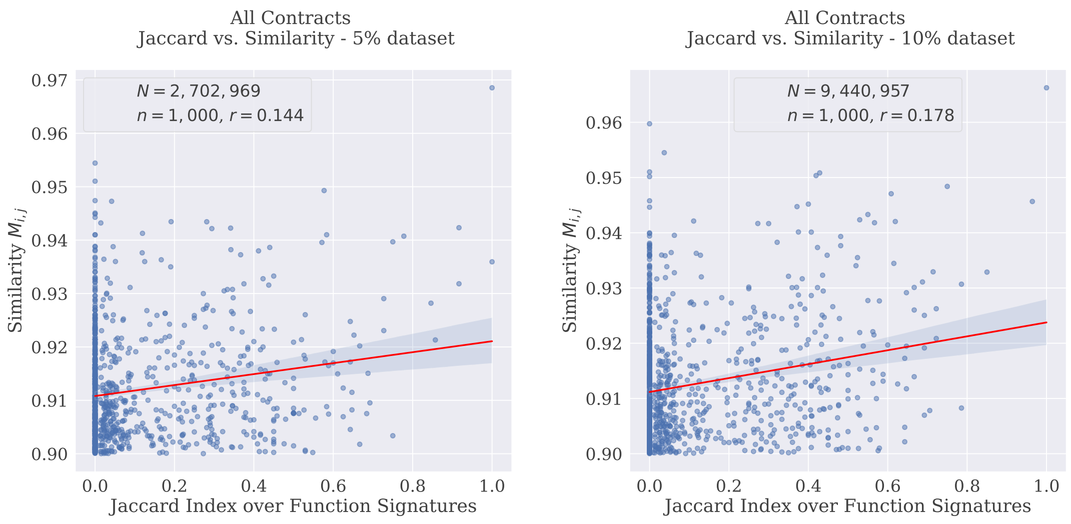


Figure 4.13: Correlation for the 5 % and 10 % datasets. The red lines represent the *line of best fit* over the sampled observations. The low correlation coefficients compared to Figure 4.11 and Figure 4.12 are explained by the homogeneity of the wallet dataset, and by the fact that wallets produce many slices on average (see Table 4.3).

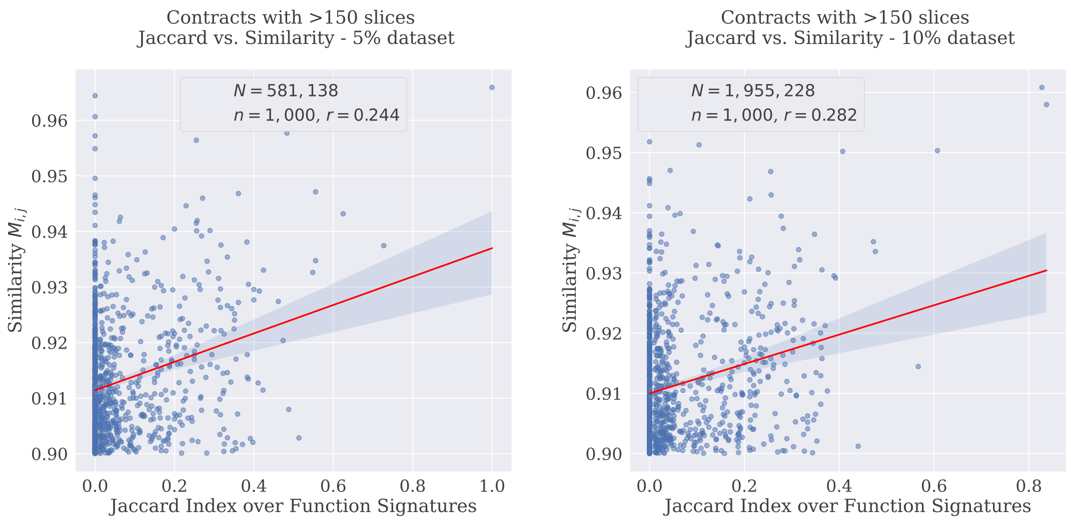


Figure 4.14: Correlation for the 10 % and 5 % datasets with limited number of slices. The red lines represent the *line of best fit* over the sampled observations. Compared to Figure 4.13, the correlation coefficient **r** is increasing when the natches are limited to those in which both contracts generate more than 150 slices.

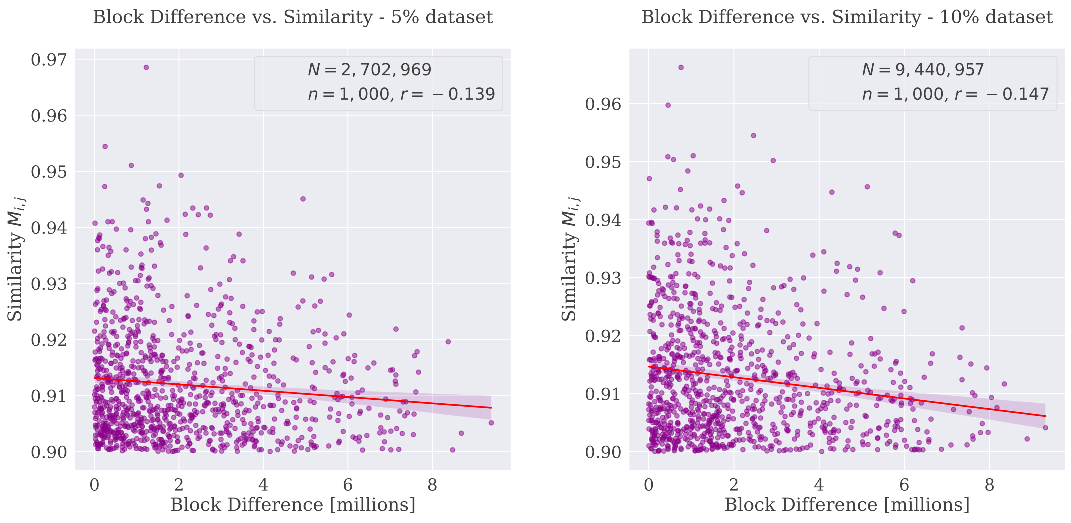


Figure 4.15: Correlation between our similarity score $M_{i,j}$ and the block difference between deployments of contract i and j. The red lines represent the *line of best fit* over the sampled observations.

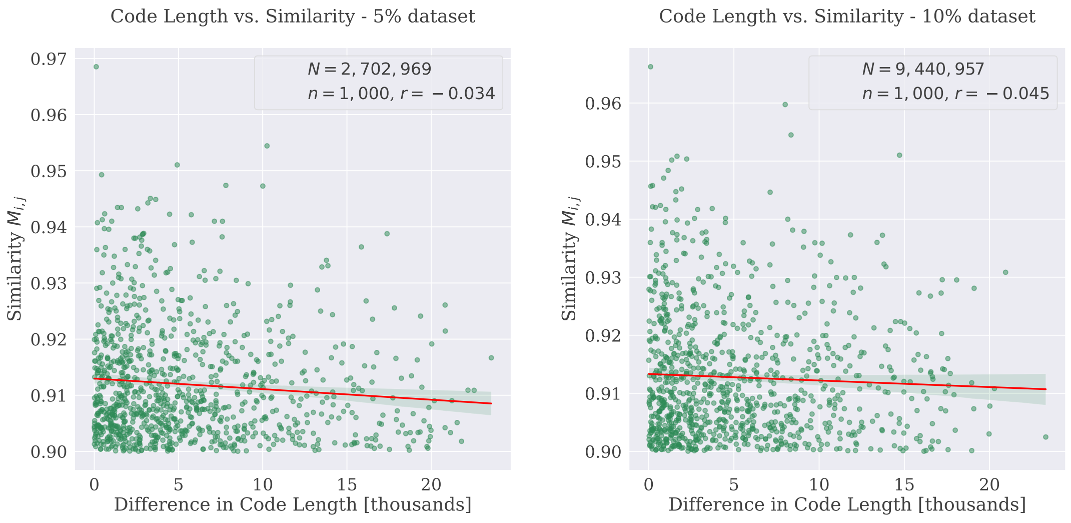


Figure 4.16: Correlation between our similarity score $M_{i,j}$ and the difference in *code length* between contract i and j. The red lines represent the *line of best fit* over the sampled observations.

CHAPTER 5

Discussion

In this chapter we discuss the results of our evaluations. We want to point out what worked well, what did not and later give our thoughts about possible applications for our approach, and what improvements we would try to make next.

5.1 1-to-1 Matching

Using the results of our experiment with three different datasets, we first want to discuss our findings in general, then draw parallels to the results of [HHY$^+$21] while highlighting key differences between their method and our version. While doing so, we also want to discuss the difficulties we experienced while reenacting their steps. Since we operate on datasets different from [HHY$^+$21], we cannot faithfully recreate their results. Rather, we focus on analysing how well *our* implementation works when used with *our* datasets. Finally, we provide a more in-depth analysis of individual matches, which helps us to identify weak points in our method and propose changes that may improve our results.

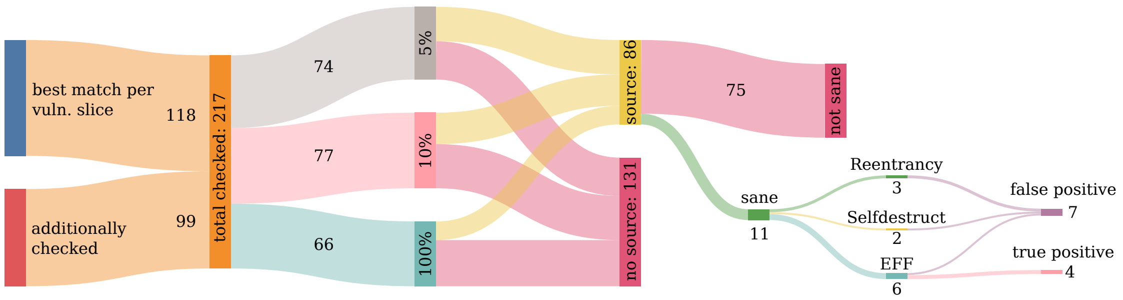


Figure 5.1: Sankey diagram consolidating the results of vulnerability matching. Of over **217** checked matches, only **11** had available source code *and* passed our sanity checks.

Figure 5.1 displays our success (or lack thereof) in finding vulnerabilities with the respective datasets. It also depicts how many of our potential matches are eliminated by performing basic sanity checks, e.g. by verifying whether a slice actually contains the operation that defines a vulnerability, or by manually inspecting if the matched slices really do resemble each other. Matches of vulnerability type *origin* and *Block-info Dependency (BID)* were disproportionally disqualified by these checks — a circumstance that is described in more detail in subsection 5.1.1.

The authors of [HHY+21] focused on five types of vulnerabilities in particular: Integer overflows, reentrancies, bad randomness, unprotected ownership and mishandled exceptions. They built one dataset containing 24 vulnerable contracts of these types and matched them against two other datasets, one containing 2 297 058 closed-source bytecodes from the blockchain, and the other containing 164 open-source vulnerable contracts from EtherScan.

We leverage the findings of Rameder et al. [RdAS22] and utilize 21 different contracts, of which we mark 44 slices as vulnerable to seven different types of vulnerabilities. Some types we use in our dataset are not directly mappable to the ones in [HHY+21], and the opposite holds true as well. We operate on three differently sized datasets of target contracts as we believe that the size of the embedding corpus has an impact on the quality of embedding vectors. Our target datasets consist of 228 793, 22 813 and 11 368 skeleton-contracts each, which are representative for 6 500 283, 56 673, and 13 118 real world contracts respectively. In contrast to [HHY+21], we do not utilize a separate dataset for validation, and rather manually validate the results of our best matches where source code is available. While the authors of [HHY+21] select their threshold $\mathbf{t} = 0.90$ trying to minimize *false-negatives* and *false-positives* within their validation set, we set it to $\mathbf{t} = 0.95$ to limit the amount of extracted data.

As they claim that they were fairly successful in finding vulnerabilities, we want to highlight deviations between our findings and theirs. For example, while they successfully detected vulnerabilities that make use of *bad randomness*, which is partly represented by our BID category, we were not able to do so. Another category for which we could not confirm any **true positive** matches is for *reentrancy* vulnerabilities. Matches for which we *were* able to do so belong exclusively to the category EFF. These are the only ones where we *think* that the method works well in finding `transfers`, which under circumstances could be exploited for DoS attacks.

We conclude that the method proposed by [HHY+21], or at least our implementation of it, works well for finding some types of vulnerabilities, but not at all for others. We were able to confirm that *our version* of their method is capable of finding slices similar to each other, based on what is extracted from the CFGs. We also observe that the *perceived* degree of similarity between matching pairs decreases proportionally to their score, albeit at different thresholds in different datasets. In order to generate good results, however, the quality of target slices, i.e. the ones containing the vulnerability, is crucial. This conclusion is supported by the fact that our initial goal of marking at least one vulnerable slice per contract resulted in matches of poor quality.

To give an example, we initially included a vulnerable contract exhibiting arithmetic bugs in our test set but were not able to accurately identify slices characteristic of the bugs. As we proceeded to target slices we deemed *near* the vulnerability, we later realized that we cannot perform meaningful sanity checks on their matches and moved on to source code analysis. Even though some manual source code inspections showed the presence of arithmetic operations that were not protected by calls to the SafeMath library, other contracts did in fact do so. Since there was no evidence that these slices do in fact embody an overflow vulnerability, we decided to exclude them from the test set. Another example for a vulnerability group that is likely as undetectable by our method are Transaction Order Dependencies (TOD).

We have repeatedly stressed the distinction between *our version* of the method compared to the one of [HHY+21]. The reason for this is due to the numerous personal interpretations and assumptions that we had to make while recreating their work, and thus, cannot confidently claim that we have produced a faithful re-implementation. To give an example for lacking detail, the original paper never explicitly mentions the criteria for terminating a slice, and implicitly only does so in a single example. Further, the authors mention the procedure of generating CFGs only twice, referring to two other works in this area but provide no other points of orientation. Another interesting finding in their work is *Table I* [HHY+21, p. 2148], in which they list instructions and corresponding tags of their output. Disregarding the fact that they explicitly mention the table's incompleteness, *Table I* contains an instruction, **CALLDATACOPY**, which, per definition, does not produce an output on the stack at all. We remain uncertain about whether this was overseen by the authors, or whether we misinterpreted the purpose of the table.

Nonetheless, we think that our implementation is on the right path to perform similarly well as the one of [HHY+21]. Retrospectively, we realize that a more conservative selection process for targeting vulnerable slices would have decreased the time to analyse, and likely would have yielded more accurate results. Additionally, a more diverse dataset of vulnerable contracts would likely help to achieve better results. This, in combination with our conclusions in the remainder of this section leads us to believe that *our version* of the method has room for improvement.

The following parts of this section give more insight into our proceedings on a per-vulnerability-type basis. We elaborate our conclusions in more detail and describe matches between slices that we believe were accurate or not. We remark that it is a difficult task to judge whether a vulnerability is present within a contract as this strongly depends on its business case. The classifications of *true positives* and *false positives* in the remainder of this section have been made based on whether we *think* that the highlighted code section can be used *contrary* to the intentions of the author of the contract.

5.1.1 Origin

The first noteworthy mention regarding Tables 4.4, 4.5 4.6 and 4.7 is that slices embodying vulnerabilities of type `ORIGIN` are overrepresented in our dataset. This fact also translates to the number of matched slices for this vulnerability type, though we want to point out that out of the **53** matches over each of the datasets, not a single one passed the sanity checks. The reason for this is quickly revealed by looking at a match of type `ORIGIN`. Listing 5.1 shows how both slices in the match are identical to each other but differ only in their initiator. Since a vulnerability of this kind cannot be present without the instruction **ORIGIN** occurring, matches like this do not pass the sanity checks.

```
{
  "191078_0x9dd8db08a907ddf82eb539bb0645d1237e9024ee_71": {
    "initiator": {"mnem": "CALLER"},
    "edges": [[0, 1], [1, 2], [2, 3], [3, 4]],
    "features": {
      "0": "AND(B, j)",
      "1": "EQ(Z, gjB)",
      "2": "ISZERO(hgjBZ)",
      "3": "ISZERO(hhgjBZ)",
      "4": "JUMPI(j, hhhgjBZ)"
    }
  },
  "1001146_0x0000000000000000000000000000vuln_QIUToken_100": {
    "target": true,
    "initiator": {"mnem": "ORIGIN"},
    "edges": [[0, 1], [1, 2], [2, 3], [3, 4]],
    "features": {
      "0": "AND(B, j)",
      "1": "EQ(Z, gjB)",
      "2": "ISZERO(hgjBZ)",
      "3": "ISZERO(hhgjBZ)",
      "4": "JUMPI(j, hhhgjBZ)"
    }
  }
}
```

Listing 5.1: Example of a match between an "`ORIGIN` vulnerability", and another slice. Note that the extracted slices are identical, and only their initiators, i.e. the slicing critera differ.

The above considerations lead us to draw our first conclusion:

Conclusion 1 Future iterations of our method should include the slicing criterion into the extracted slices as we think it would increase the quality of matches. See subsection 6.1.1 for details.

5.1.2 Selfdestruct

Out of 12 matches across three datasets for this group, only two[1,2] pass our sanity checks. While for both of these contracts source code is available, and both implement self-destruction, we note that the functionality is protected by `onlyOwner` modifiers. Thus, these matches fall under the category **false positives**. We think that the reason for this is two-fold and also observable with other vulnerability types: On the one hand, a slice that comprises the **SELFDESTRUCT** operation will not necessarily include logic that is executed at the beginning of a function, e.g. by a modifier. This means that, depending on the size of the functions body, code implementing access control logic is unlikely to be included in a slice embodying a vulnerability. On the other hand, even if such logic was included in the extracted slices, what makes this vulnerability threatening is the absence of such logic — not the presence thereof. Including such modifier logic into a slice would most likely not find any more vulnerabilities but other contracts that protect their self-destruction using such a modifier. Based on these thoughts, we make our second conclusion:

Conclusion 2 Since the use of modifiers is widespread in solidity, a way to improve our process could be to ensure the absence of such before extracting a slice as vulnerable. As access control modifiers often implement only few lines code performing similar kinds of checks, we think it could be viable to identify typical slices corresponding to such code, and in addition to checking for the presence of a vulnerable slice, check for the absence of such modifier slices in the matched contract where applicable.

5.1.3 Externally Forced Fail

As depicted in Table 5.1, out of 8 matches, 6 passed our sanity tests. Of those six contracts, only one contract's source code was available, while none of the others' could be obtained, neither directly, nor by association over their respective skeleton group. Interestingly, contracts with the same skeleton as the last contract in Table 5.1 had been deployed 95 times from five different accounts, but for none we could obtain the source code.

We looked into the single contract stemming from the 5 % dataset whose source code was available[3], and technically we were indeed able to identify a vulnerability of type *EFF*. Listing 5.2 depicts a shortened version of this contract.

In particular, the functions `sowCorn` and `reap` are vulnerable to EFFs due to the fact that their caller must specify the target contract's address himself. Since no restrictions to the address exist, an attacker could pass arbitrary contracts under his control as arguments for these functions, including ones that fail forcefully. As no other logic is executed besides the two function calls, we conclude that even if the contract is theoretically prone to EFFs, this particular instance is not a contract that can be meaningfully exploited by

[1] `0x9f751aaacc74e55a27a19419c332e02aa96ed961`
[2] `0x738dfaf60910ebcb4cd369cb983b5d36467e9673`
[3] `0x34ea8cdc7837d3a84f5869909104bdb1a7c8cb35`

an attacker. Lastly, we note that the `onlyParent` modifier in line 20 remains unused within the contract, and think that this was either overseen or — given the name of the contract — done intentionally by the creator. We classify this match as a **true positive**.

```
1  pragma solidity ^0.4.19;
2
3  interface CornFarm
4  {
5      function buyObject(address _beneficiary) public payable;
6  }
7
8  interface Corn
9  {
10     function balanceOf(address who) public view returns (uint256);
11     function transfer(address to, uint256 value) public returns (bool);
12 }
13
14 contract howbadlycouldthisgowrong {
15     address public parentAddress;
16     event ForwarderDeposited(address from, uint value, bytes data);
17
18     /* constructor and fallback function omitted */
19
20     modifier onlyParent {
21         if (msg.sender != parentAddress) {
22             revert();
23         }
24         _;
25     }
26
27     address public farmer = 0xC4C6328405F00Fa4a93715D2349f76DF0c7E8b79;
28
29     function sowCorn(address soil, uint8 seeds) external {
30         for(uint8 i = 0; i < seeds; ++i){
31             CornFarm(soil).buyObject(this);
32         }
33     }
34
35     function reap(address corn) external{
36         Corn(corn).transfer(farmer, Corn(corn).balanceOf(this));
37     }
38 }
```

Listing 5.2: Contract `howbadlycouldthisgowrong`. The highlighted lines mark code that may result in EFFs, though the (malicious) caller would be the only one damaged when trying to exploit this contract.

After this arguable initial success, we decided to investigate further matches for the target slice matching with this contract. The 19 next best matches were looked at, all of which passed our sanity checks. For five of those contracts we were able to obtain source codes and proceeded to make manual inspections.

skeleton-contract	**dataset**	**sane**	**source**	**represented contracts**
0x34ea8cdc7837d3a84f5869909104bdb1a7c8cb35	5 %	yes	yes	1
0x0347cd66ea7756377028e494e92845c800ee1521	5 %	yes	no	1
0xff93908c8e92181d623f4a58bceb5bf53fb143c5	5 %	no	-	-
0xf547229a3b21c525630eda4fa334fada82464358	10 %	yes	no	1
0xacdb43d57fbea59d7aa4e9e6fd274ea78d0610cb	10 %	yes	no	1
0x20d14e391a80dfa8e28778c263e41e780fb8f4b8	10 %	no	-	-
0x8b48cb5d71ae681a5fbba2064a330afbc448aaa5	100 %	yes	no	1
0x2ca103f6c1b5bdc36118c05491eea85080e93d14	100 %	yes	no	95

Table 5.1: Matches for type EFF. The matches in this table represent the best matches per vulnerable slice and dataset that exceed $t = 0.95$.

ListingsERC20 [4] This contract is a hybrid token / auctioning contract where a seller could create an *unsellable* listing by creating a contract that fails when its `transfer` function is called. Listing 5.3 displays the vulnerable function, with the unsafe transfer taking place on line 16. Since exploiting this could somewhat deny service to others, we classify this match as a **true positive**.

```
 1 function buyListing(bytes32 listingId, uint256 amount) external payable {
 2     Listing storage listing = listings[listingId];
 3     address seller = listing.seller;
 4     address contractAddress = listing.tokenContractAddress;
 5     uint256 price = listing.price;
 6     uint256 sale = price.mul(amount);
 7     uint256 allowance = listing.allowance;
 8     require(now <= listing.dateEnds);
 9     require(allowance - sold[listingId] >= amount);
10     require(allowance - amount >= 0);
11     require(getBalance(contractAddress, seller) >= allowance);
12     require(getAllowance(contractAddress, seller, this) <= allowance);
13     require(msg.value == sale);
14     ERC20 tokenContract = ERC20(contractAddress);
15     require(tokenContract.transferFrom(seller, msg.sender, amount));
16     seller.transfer(sale - (sale.mul(ownerPercentage).div(10000)));
17     sold[listingId] = allowance.sub(amount);
18     ListingBought(listingId, contractAddress, price, amount, now, msg.sender);
19 }
```

Listing 5.3: A vulnerable contract where the seller of a listing could always force a fail to deny buyers their purchase.

DMToken [5] This contract exhibits an unchecked return value on line 261. While an EFF vulnerability *theoretically* exists, it is highly unlikely that this can ever be exploited unless the "upgradeMaster" loses control of the target contract. Thus, we classify this match as a **false positive**.

[4] `0xab24cd33766da327ecd4ec9e46e2e7ba72cda783`
[5] `0x5c751a3a3375a97463a4b5f000c3f700802e903a`

Draw [6] Here, an EFF vulnerability exists, though an attacker would not be able to exploit this contract in the sense of stealing ether. Parts of the contract are displayed in Listing 5.4. The vulnerability is constituted by line 27, where an "attacker" could be participating from a contract that forces upon being called with the `transfer` function. For this reason, we classify this match as a **true positive**.

```
 1  /* pragma and Ownable contract omitted*/
 2  contract Draw is Ownable {
 3
 4      /* storage variable declarations, constructor, initiator function and
            fallback function redirecting to joinGame omitted */
 5
 6      function joinGame() public payable {
 7          require(msg.sender != owner);
 8          require(msg.value == 100 finney);
 9          require(counter < MAX_PLAYERS);
10
11          players[counter] = msg.sender;
12          counter++;
13          slots_left = MAX_PLAYERS - counter;
14
15          if (counter >= MAX_PLAYERS) {
16              last_winner = endGame();
17          }
18      }
19
20      function endGame() internal returns (address winner) {
21          require(this.balance - owner_balance >= 900 finney);
22          tdelta = now - t0;
23          index = uint(tdelta % MAX_PLAYERS);
24          t0 = now;
25          winner = players[index];
26          initGame();
27          winner.transfer(855 finney);
28          owner_balance = owner_balance + 45 finney;
29      }
30      /* balance query and withdrawing functions omitted */
31  }
```

Listing 5.4: A player using a contract (which does not implement `receive` or `fallback`) to participate would be denied his winnings.

Forwarder [7] This contract uses a `if !call then revert()` instead of a `require(call)` construct, which might be reason that this contract was matched. Since the vulnerability does in fact not exist, we claim this to be a **false positive**.

Snake [8] Our last match is a contract that is definitely vulnerable to an EFF. The contract works by buying parts of a snake off of a previous owner. Everytime the head of the snake is traded, its body grows by a length of one. An attack scenario could comprise buying any part of the snake for its current price using a malicious contract that implements a `transfer` function which unconditionally reverts. This way, an attacker could deny potential buyers the ability to buy the snake part from

[6] `0xc32c4bd955cfd68bddbc13b4baef73bcef0e09da`
[7] `0xed8d3b7221453777f67622f5a4fea8e1b427d517`
[8] `0x01dd8186b8f38dfa01ea2c044355ea95206a4481`

him. Applying such an attack to the snake's head would additionally prevent the snake from growing any larger. Both of these attack scenarios are EFFs that lead to a DoS. Listing 5.5 depicts the vulnerable contract's source code, which we classify as a **true positive**.

```
pragma solidity ^0.4.19;

contract Snake {
    address public ownerAddress;
    uint256 public length; // stores length of the snake

    mapping (uint256 => uint256) public snake;
    mapping (uint256 => address) public owners;
    mapping (uint256 => uint256) public stamps;

    event Sale(address owner, uint256 profit, uint256 stamp); // 'stores' sales
        of tokens

    function Snake() public {
        ownerAddress = msg.sender;
        length = 0; // set initial length of the snake to 0
        _extend(length); // create head of the snake
    }

    // called when someone buys a token from someone else
    function buy(uint256 id) external payable {
        require(snake[id] > 0); // must be a valid token
        require(msg.value >= snake[id] / 100 * 150);
        address owner = owners[id];
        uint256 amount = snake[id];

        snake[id] = amount / 100 * 150; // set new price of token
        owners[id] = msg.sender; // set new owner of token
        stamps[id] = uint256(now);

        owner.transfer(amount / 100 * 125);     // transfer gain
        Sale(owner, amount, uint256(now));
        // if this is the head token being traded:
        if (id == 0) {
            length++; // increase the length of the snake
            _extend(length); // create new token
        }
        ownerAddress.transfer(this.balance); // transfer to owner
    }
    // increases length of the snake
    function _extend(uint256 id) internal {
        snake[id] = 1 * 10**16;
        owners[id] = msg.sender;
    }
}
```

Listing 5.5: A contract where DoS is possible against all participants.

A proof of concept (PoC) exploit contract might look as follows.

```
pragma solidity ^0.4.19;
contract SnakeEater {
    Snake public snake = Snake(0xd9145CCE52D386f254917e481eB44e9943F39138);

    function buyPart(uint part) payable public{
        snake.buy.value(msg.value)(part);
    }
}
```

Listing 5.6: PoC exploit for contract `Snake`.

A contract that is called via a `transfer` which implements neither of `receive` or `fallback` will fail. Thus, it suffices to buy a snake's part using a single-function contract as depicted in Listing 5.6. Listing 5.7 contains the slice that matched with the contracts listed for this category.

```
{
    "initiator": {
      "mnem": "SLOAD"
    },
    "edges": [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]],
    "features": {
      "0": "EXP(j, DZ)",
      "1": "DIV(fjDZ, Y)",
      "2": "AND(ffjDZY, j)",
      "3": "AND(gjffjDZY, j)",
      "4": "CALL(j, gjgjffjDZY, fhjX, ij, fijij, ij, j)",
      "5": "ISZERO(gjgjffjDZY)",
      "6": "ISZERO(hgjgjffjDZY)",
      "7": "JUMPI(j, hhgjgjffjDZY)"
    }
}
```

Listing 5.7: The slice that matched with the named contracts.

Conclusion 3: We conclude that our method at least somehow works for contracts that neglect checking the return value of transfers. We take these findings into consideration later.

5.1.4 Delegatecall

Since none of the matches shown in Tables 4.5, 4.6 and 4.7 incorporated a slice containing the instruction **DELEGATECALL**, and thus passed our sanity checks, we decided to extend our search trying to find a well-matched slice. Unfortunately, even after checking 30 additional slices, there was not a single one fulfilling the above requirement. Since we ordered the matches by their score and checked them in descending order, the matches were getting more dissimilar as we advanced through the list. We try to explain this behaviour by displaying the slice that we marked as vulnerable in Listing 5.8.

```
{
    "target": true,
    "edges": [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]],
    "features": {
      "0": "EXP(j, DZ)",
      "1": "DIV(fjDZ, Y)",
      "2": "AND(ffjDZY, j)",
      "3": "AND(gjffjDZY, j)",
      "4": "AND(gjgjffjDZY, j)",
      "5": "MSTORE(fjij, gjgjgjffjDZY)",
      "6": "SUB(gjffjDZY, ij)",
      "7": "DELEGATECALL(X, V, ij, fgjffjDZYij, ij, j)"
    }
}
```

Listing 5.8: While our target instruction included **DELEGATECALL** as the very last instruction, none of the matched slices did the same.

While many of our inspected slices where *almost* identical to the target slice, we noticed that the last instruction in our target slice — the instruction defining this vulnerability — was missing in all of them. We believe that this may be due to the fact that the instruction in question was the very last one, and that such instructions generally perform worse in the matching than ones occurring earlier in the slice. This might be ascribed to the way how graph2vec uses *sub-graphs* as context for its labels. Since the neighbouring sub-graphs of the last instruction, i.e. of `DELEGATECALL` contain very little nodes, we think that graph2vec may not have been as successful in encoding this instruction into the vector space as it was for earlier occurring ones.

Conclusion 4 We think this behaviour could be mitigated by introducing bidirectional edges instead of ordinary ones. This may seem counter-intuitive at first, as such edges would go against the sequence direction of the program. However, we think it could help when dealing with situations like we encountered here.

5.1.5 Reentrancy

Of 38 matches over all three datasets that were individually looked at, 20 passed the sanity checks. However, only three of those had their source code available. In the first match[9], in the function `executeOperation`, reentrancy is theoretically possible through a function call. However, the code can de facto be ruled out to ever be exploited due to the fact that the reentering contract would be under the control of the owner. Thus, we classify this as a **false positive**.

The second match[10] is classified as a **false positive** as well due to the fact that a `transfer` rather than a call is used. This limits the amount of available gas for the

[9] 0x1597e31e4831284a63ff9e98faa70b16612a1ac8
[10] 0xca30a6938d8a2c70c547a694755bf6d81e04b2ea

receiver to 2 300, and was specifically introduced to mitigate reentrancy attacks.

The last match[11] is another **false positive** due to the fact that a special `nonReentrant` modifier is used on functions, preventing reentrancy. While the original authors were successful in detecting reentrancy vulnerabilities, we conclude that it is rather difficult to find instances where theoretical reentrancies can really be exploited.

5.1.6 BlockInfoDependency

Generally speaking, matches of this kind were of low quality. While subjectively good matches for other vulnerability types were observed to be around a threshold of ~0.99 for the sampled datasets, and around ~0.97 for the full dataset, we observed matches for this kind of vulnerability generally to be way below those thresholds. The reason for this likely stems from causes that we have already mentioned in subsection 5.1.1.

```
{
    "target": true,
    "initiator": {"mnem": "BLOCKHASH"},
    "edges": [[0, 1], [1, 2]],
    "features": {
      "0": "DIV(j, CZ)",
      "1": "ADD(Y, fjCZ)",
      "2": "ADD(X, ffjCZY)"
    }
}
```

Listing 5.9: Example slice for a vulnerable slice with a **BLOCKHASH** criterion.

As observable in Listing 5.9, the only information within the slice that a block info instruction was used is the group symbol of **BLOCKHASH**, **C**, which it shares with 10 other instructions (see Table 3.2). We think this is a very vague encoding of information, and again recommend including the slicing criterion to the slice itself, which may result in a more *characteristic* target slice and may lead to matches of better quality.

5.1.7 UncheckedReturnValue

While two out of twelve matches pass the sanity checks, there is no source code available for the contracts on Etherscan.io.

[11] `0x08c96398c00c0b890da21ebdd26c67487455d2bf`

5.2 N-to-M Matching

In this section we discuss the results of our experiments concerning *N-to-M* matching (contract matching).

5.2.1 Standard Contracts

As mentioned in subsection 4.4.1, we focus on two subsets of matches from the dataset **10 % sample ∪ standard contracts** for this experiment: One subset containing matches between our synthetically generated contracts on both sides of the match, and the other containing matches between synthetic and real contracts. Using the numbers of plausible and implausible matches in Tables 4.11, 4.12 and Figures 4.3a, 4.3b we will proceed to draw our conclusions. We want to point out that we omit discussing results for the 5 % dataset due them exhibiting little deviations compared to the 10 % dataset.

First, we want to revisit our definition of *plausible* and *implausible* matches using the contract `ERC721MintableBurnableImpl` as an example. This contract inherits from three interfaces: `ERC721`, `Mintable` and `Burnable`. We define a match as *plausible* if there is at least one interface which both contracts in the match inherit from, and implausible otherwise. In this case, the second contract in the match would have to implement any of the three interfaces mentioned above in order for the match to be ruled as plausible. We want to stress the fact that a *plausible* match does not guarantee that two contracts were matched because of the interface that they are sharing, just the same as an *implausible* match does not necessarily mean that there are no similarities between the matched contracts at all. Knowing this, we want to highlight our findings using both of the subsets separately before drawing our conclusions.

Synthetic Subset

Generally speaking, we can observe that our method performs moderately well for matches above the threshold $\mathbf{t = 0.95}$. Depending on the contract, however, the ratio of plausible matches relative to the total matches deviates largely, and in an unexpected manner in one instance. The contract that stands out as negative outlier in this regard is `ERC20`. Compared to other contracts, it registers an *increase* in the percentage of plausible matches as the threshold is decreased. This effect seems counter-intuitive at first, and may be explained by the fact that there are four other synthetic contracts inheriting this interface, making this interface overrepresented. Out of 786 implausible matches for contract `ERC20`, 517 can be attributed to variations of the `ERC721` contract and 268 to the `ERC1155` family.

Contracts that performed rather poorly are `ERC1820Implementer` and `SafeMathMock`, which is unsurprising given that both contracts only generate one slice. For `SafeMathMock`, the low number of plausible matches is also explained by the fact that ordinary implementations of the ERC-suite do generally not expose their functions performing SafeMath operations, which makes detecting plausible matches impossible using our method.

Another observation we make is that matches with the contract `ERC777Token` have a 100 % rate of being plausible within our synthetic subset, regardless of the threshold t. Despite the fact that the contract in our synthetic setup implements both interfaces, `ERC20` and `ERC777`, none of the plausible matches in this category are *cross-matches*, i.e. none of the matched contracts are such that implement `ERC20` but no other interface. This means that in all matches with this specific contract, both of the contracts implement both interfaces. The reason for this may be due to a few very long and characteristic slices that were extracted from our `ERC777Token` contracts.

Real Subset

Looking at matches between synthetic and real contracts, we observe that our method performs rather poorly, with the best two sticking out as contracts `ERC20BurnableMock` and `ERC20Mock`. Interestingly enough, the contract that performed worst within synthetic matches, `ERC20`, is among the better performing ones when matched against real contracts. Contracts that generate many matches with a high threshold of $t = 0.95$ are `ERC20MintableMock` (8 735 plausible), `ERC165Mock` (8 645 plausible), and `ERC721Mock` (3 976 plausible).

Our intuition is that this experiment performed badly due to the fact that contracts in the ERC-suite generally generate a small amount of slices. This was also the reason why we decided to look at the similarity of matches in only one direction, i.e. whether an ERC contract is similar to the match, but not the other way around. We believe the reason for the low number of slices to be that the reference implementations we obtained from OpenZeppelin are *minimal* implementations with little to no program logic, whereas real ERC token contracts generally comprise more business logic, and therefore generate more slices.

The conclusion we draw from this experiment is that the number of generated slices are key when trying to relate contracts to each other. This, in combination with the *uniqueness* of extracted slices determines the quality of our matches.

Lastly, we want to point out that there were no noteworthy differences in the number of plausible matches between the larger and the smaller dataset.

5.2.2 Wallet Contracts

As the nature of the results does not deviate noticeably between the two datasets, we want to focus on discussing results using the 10 % for this experiment as well. We will draw a more high level comparison between the datasets at the end of this subsection.

Compared to our experiment using standard contracts, matching wallets with each other seems to have been more successful at first glance. By looking at Table 4.14, we can observe that our matching process works well for classifying different wallet types given a sufficiently high threshold. Figure 4.8 and Table 4.14 show that by choosing a high value for $\mathbf{t}$, we are able to reduce mismatches to $\mathbf{0}$, albeit at the price of a low number of overall matches. We take this as an indicator in favor of our method, and that it *can* work as intended given the right circumstances. Decreasing the threshold to $\mathbf{t = 0.92}$, we can observe that for a total of 20 614 matches, 20 481 are between the same types, and 153 between different types of wallets. Here, it turns out that wallets of type "consumer wallet" are prone to mismatch with type "multisig Stefan George" — a circumstance that manifests even stronger when further decreasing the threshold. At a threshold of $t = 0.90$, we observe that many of the mismatches can be attributed to the group "multisig Stefan George". This is to be expected to some extent, as this group is by far the largest group of wallets, as depicted in Table 4.16. The wallet types listed in Table 5.2 are wallet type pairs prone to mismatch in absolute numbers.

Wallet Type 1	**Wallet Type 2**	**Mismatches**[1]
multisig Stefan George	multisig Gavin Wood/Ethereum/Parity	2 699
multisig Stefan George	smart GnosisSafe	990
multisig Stefan George	consumer wallet	742
multisig Stefan George	multisig WalletSimple/BitGo	516

[1] In this table we count mismatches between two wallet groups regardless of their direction.

Table 5.2: Common Wallet mismatches.

We note that our threshold $t = 0.90$ was likely too high to capture all possible matches. This circumstance is depicted by Tables 4.15 and 4.16, where large differences between values of column t-sum(n) and the actual number of matches are observable. However, decreasing the threshold to capture more matches is not a viable strategy as the number of mismatches rapidly increases for thresholds below $t = 0.92$. Rather, we think that adjusting the method according to our recommendations in chapter 6 is a more promising approach.

Particularly interesting are the first mismatches that occur when decreasing the threshold from $t = 0.95$, as well as mismatches that display a high *Jaccard index* over function signatures. In the remainder of our discussion we will look at a few of such mismatches, and reason about why they might have happened.

The first mismatch with source codes available for both wallet contracts is one between type "multisig WalletSimple/BitGo forwarder"[12] and type "spendable wallet"[13]. The most obvious overlap between their functionalities is that both contracts implement code to withdraw ether to an address that is persisted in the storage, and to transfer tokens of type `ERC20` using a caller-specified token contract. Our investigations have shown that the *Forwarder* contract is used in combination with so-called *Proxy* contracts, which delegate calls to said *Forwarder* to make use of its implementations. This circumstance,

[12] `0xf5e967b72f20892f4d28fce0fc0f99c9898f3c57`
[13] `0x36d76772c416bab17661e1596e023f1a04d6bef0`

however, is out of scope of our similarity matching and should only be registered as a side note.

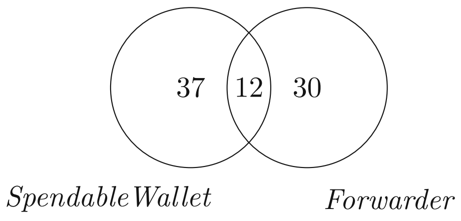


Figure 5.2: Number of slices per contract in the first wallet (mis)match. The intersection represents the set of slices that occur in both sides of the match.

Figure 5.2 displays that these contracts share 12 identical slices. While the remaining slices show resemblances as well, we cannot tell with certainty which slice pairs were matched as this information is dismissed in the extraction process. Listings 5.10 and 5.11 are excerpts of these contracts that show functionally similar code, which probably lead to the 12 shared slices.

```
function flushTokens(address tokenContractAddress) external onlyParent {
  ERC20Interface instance = ERC20Interface(tokenContractAddress);
  address forwarderAddress = address(this);
  uint256 forwarderBalance = instance.balanceOf(forwarderAddress);
  if (forwarderBalance == 0) {
    return;
  }

  require(
    instance.transfer(parentAddress, forwarderBalance),
    "Token flush failed"
  );
}

function flush() public {
  uint256 value = address(this).balance;
  if (value == 0) {
    return;
  }
  (bool success, ) = parentAddress.call{ value: value }("");
  require(success, "Flush failed");
  emit ForwarderDeposited(msg.sender, value, msg.data);
}
```

Listing 5.10: Two functions of the *Forwarder* wallet, `flushTokens` and `flush`.

```
function claimTokens(address _token) public onlyOwner {
  if (_token == 0x0) {
    owner.transfer(address(this).balance);
    return;
  }

  ERC20 erc20token = ERC20(_token);
  uint256 balance = erc20token.balanceOf(address(this));
  erc20token.transfer(owner, balance);
  emit ClaimedTokens(_token, owner, balance);
}
```

Listing 5.11: Function `claimTokens` of the *SpendableWallet* wallet.

We point out that the same *SpendableWallet* contract matches with two other *Forwarder* contracts[14,15], both of which slightly vary from the previous one, but in essence implement the same functionalities. Judging from this circumstance, we think a valid conclusion is that mismatches are consistent in their causality. Further, we note that there is at least one other *SpendableWallet* contract[16] (mis)matching with the same three *Forwarder* wallets above. The two *SpendableWallets* differ insofar as the second one encloses its `transfer` statements with `require` logic.

The third type of mismatch for which source code is available is between the groups "smart GnosisSafe"[17] and "multisig Stefan George"[18]. Due to the fact that these contracts span several hundreds of lines, manual source code analysis becomes tedious and unreliable. We note that there are a few similar functionalities such as different types of signature validation, but we are not able to pinpoint the exact sections of codes that these contracts share among each other, if any.

By ordering mismatches by the highest Jaccard index we discover a mismatch between wallets of types "simple wallet 2"[19] and "spendable wallet"[20] Based on their source codes, there are few similarities between these two contracts, other than that they both inherit from the same *Ownable* contract. Due to the fact that the first contract in this mismatch does only include two basic, single-line functions, we have reason to believe that these contracts matched for the sole reason of inheriting a common contract.

This intuition is further reinforced by the fact that the next mismatch in our list is between the same "simple wallet 2" as before, but this time with another "spendable wallet"[21]. Again it is the case that the only shared common code is the inherited *Ownable* contract, but simply due to the fact that there is little else to share.

To conclude this subsection, we want to glance at the performance of our method when compared with the *Jaccard index* over function signatures. As seen in Figures 4.11 and 4.12, our similarity scores show a moderate correlation with the Jaccard index with a Pearson's coefficient of $r = 0.643$ and $r = 0.607$ respectively. Since wallet contracts were predominantly classified by their function signatures it comes as no surprise that the marginal plots in either of the Figures show that the Jaccard index is a much better discriminator between matches and mismatches than our similarity metric. Nonetheless, we think that the experiments conducted with wallet contracts are ones where our method shows its strengths.

[14] `0x85e0e3464bbb9a75d6b3dd4db241e49721fda555`
[15] `0x6a92fad4231183104ad67fa17857030d576a99eb`
[16] `0x016fa0e04822bdaadea2c8335198482b1948d5c4`
[17] `0xb6029ea3b2c51d09a50b53ca8012feeb05bda35a`
[18] `0x4cadb4bad0e2a49cc5d6ce26d8628c8f451da346`
[19] `0x308b773f33e3f1a38ada8928c3f6dc2d8861d573`
[20] `0x016fa0e04822bdaadea2c8335198482b1948d5c4`
[21] `0x2254f46dedafa2a03f59008456a7400cfadcaf73`

5.2.3 Correlation Metrics

We conclude this section by discussing our hypotheses from section 3.6.

Hypothesis 1: *A higher Jaccard index J between two contracts i and j generally implies a higher similarity score $M_{i,j}$. A larger dataset is more likely to exhibit this behaviour.*

Figure 4.13 displays a slight correlation between our matching score and the Jaccard index, albeit much lower than it was for wallets in Figures 4.11 and 4.12. We explain the difference in correlation by the fact that the wallet datasets consist of contracts that on the one hand create more slices per contract, and on the other form a more homogeneous corpus. After having made this observation, we limited results to matches where both contracts generate over 150 slices. As expected, matches of this kind display a higher correlation. We also confirm our intuition that contracts trained in the larger dataset display a slightly higher correlation between the two metrics — an effect that is observable in either of the measurements. Interestingly, this trend is reversed when regarding the experiments containing wallet contracts as depicted by Figure 4.11 and Figure 4.12.

Hypothesis 2: *If two contracts have the same creator, they are more likely to be similar than two contracts with distinct creators. A larger dataset is more likely to exhibit this behaviour.*

According to the classification made by Cohen et al. (listed in subsection 4.4.3), there is small correlation between the two metrics for thresholds higher than $t = 0.93$. Correlations for thresholds smaller than $t = 0.93$ turn out to be negligible. The highest observed value for η^2 in Table 4.17 was 0.0577 using the 10 % dataset and a threshold of $t = 0.95$.

Hypothesis 3: *Contracts with low block difference between their deployments, are more likely to be similar. A larger dataset is more likely to exhibit this behaviour.*

According to Figure 4.15, the general intuition of a negative correlation between block difference and similarity holds true. The effect may be small, but is still detectable as indicated by the sign of the correlation coefficients $r = -0.139$ and $r = -0.147$ for the 5 % and 10 % dataset, respectively. We note that the correlation for the 10 % dataset is *marginally* stronger.

A third correlating factor could be the solc version with which the contracts were compiled. We think it is reasonable to assume that contracts whose deployments are separated by millions of blocks tend to also be compiled using compiler versions farther apart than contracts that were deployed in quick succession. An investigation whether this claim holds true is left for future work.

Hypothesis 4: *Contracts with low difference in code length are more likely to be similar. A larger dataset is more likely to exhibit this behaviour.*

While the directions of the correlation coefficients align with our intuition, their magnitudes are negligible. Figure 4.16 depicts this circumstance with a correlation coefficient of $r = -0.034$ for the 5 % dataset, and $r = -0.045$ for the 10 % dataset.

5.3 Research Questions

Finally, we revisit our research questions from chapter 1.

RQ$_1$: Can we verify that the method of [HHY+21] performs as described when used on our dataset?

During our attempt to reimplement the approach of [HHY+21], we had to make many assumptions and design choices ourselves due to lacking details in their description. Hence, our implementation deviates to an unknown extent from the original. We can therefore not verify their exact method. However, we succeeded in implementing a prototype, and showed that it can be used as intended in section 5.1. Our method having repeatedly detected one particular type of vulnerabilities — Externally Forced Fail (EFF) — speaks in favor of our method, whereas our observations for other vulnerability types investigated by us do not. Given the numerous weak points and false positives detected in section 5.1, we believe that a reassessment of our method needs to be made after implementing the proposed improvements. Nonetheless, we think that the key functionality, matching similar slices to each other, is provided by our method.

RQ$_2$: How well can semantic similarities between smart contracts be detected by extending the method of [HHY+21]?

As we have seen with some matches in the previous section, we deem it possible that our method can find similarities between contracts in *some cases*, although it does so with limited precision. Given a high enough similarity score, we believe that our method is also able to detect similarities between wallets of different groups. For instance, we have shown that a mismatch between two wallet groups occurred because of two contracts extending the same base contract. Depending on the experiment, we were also able to show small to moderately large correlation between our similarity metric and the Jaccard index over function signatures of two contracts. Furthermore, we were able to confirm our hypotheses which correlate our similarity score with various other features of contract pairs. However, we have also shown that our method is limited when e.g. trying to find contracts implementing standard interfaces, where a low number of extracted slices worked against our method. Again, we think a reassessment is necessary after implementing the proposed improvements.

CHAPTER 6

Conclusion

6.1 Future Work

This section lists our suggestions for improving our method, ordered by the presumed impact we believe they will have.

6.1.1 Adding the Criterion to the Slice

While [HHY+21] do not explicitly mention whether they include the criterion to the slice, we opted against doing so during implementation. The clue that lead us to make this decision is the fact that their provided example does not depict the inclusion of the slicing criterion. Retrospectively, we believe that doing so would likely have reduced the number of contracts failing our sanity checks in section 5.2. For this reason, we propose that future iterations of our method should include the slicing criterion as first label in the extracted slice / graph. We think that especially slices with a small number of commonly occurring instructions (e.g. Figure 3.5a) could benefit from having an additional node with the criterion as label.

6.1.2 Optimal Algorithm as TensorFlow function

In this work we have investigated two possible methods for *n-to-m* matching, one yielding an optimal solution using the CPU and the other being more performant in terms of throughput using the GPU. Since other algorithms for finding the optimal solution for the assignment problem exist, we think trying to implement those using the TensorFlow framework and its *tf.autograph* module could be a viable option. One suitable candidate algorithm for such an implementation is the Hungarian Algorithm as described by Kuhn et al. [Kuh55]. We think the operations used in this algorithm may be well-supported by TensorFlow.

6.1.3 Adjust the Matching Metric

In our experiments, we realized that the number of slices extracted from contracts in a match have a large impact on its quality. In a worst case scenario, a contract for which a single slice was extracted may have a high matching score with a contract, for which 150 slices were extracted. We propose that further iterations of our scoring metric should account for this difference by e.g. applying a penalty to scores where the slice count of contracts differ largely.

6.1.4 Hyperparameter Optimization

There are several parameters for which further optimizations could be performed in our process. This subsection lists a few of them.

Criteria Set

As the focus of [HHY$^+$21] is on the detection of vulnerabilities, their set of criteria that initiate slicing consisted of instructions that introduce data originating from *outside* the blockchain. This criteria set originally consisted of only 7 instructions, which we extended for our uses as depicted by Table 3.1. It is reasonable to assume that this set is incomplete in the sense that by adding or removing certain criteria one could achieve better results.

Stack Configuration

We believe the method of using a static stack configuration rather than calculating a realistic one yielded great benefits to our process, primarily to its runtime. As our algorithm currently finishes slicing procedure once an out-of-bounds element from the stack is accessed, increasing the number of elements in the static stack configuration could be a simple way to extract longer and more characteristic slices.

Varying the Graph Limits

Since traversing the CFG is expensive for larger graphs, we had to limit the depth until which we do so. Foremost, we limited the depth of our graph recursion to 10, and within the same recursion never traverse the same edge twice, even if it would be part of another path. While this was done in order to keep computation times low, we also tried to limit the length of slices this way while aiming to increase their numbers thinking that this would yield better matching results.

graph2vec Parameters

In an attempt to stay as true to [HHY$^+$21] as possible (and to limit runtime) we assumed their embedding size of choice (64), but also other default parameters of graph2vec and omitted hyperparameter optimization altogether. However, since the optimal choice of the embedding size is largely dependent on the used dataset, we retrospectively think that spending effort to do so could lead to meaningful improvements of the method, especially as graph2vec requires several parameters that could be optimized for.

6.1.5 Improved Dataset of Vulnerable Contracts

By crafting the dataset more carefully, and by using a validation dataset like the authors did in [HHY+21], we think that more reliable statements can be made about the quality of our findings. Another addition could be to consider contracts written for Solidity versions smaller than 0.4 as compilers of early versions lack security relevant features.

6.1.6 Apply graph clustering

Since N-dimensional embedding vectors allow for more than just comparing the angle between them, we think that it may be a viable option to perform clustering on vectors. As Narayanan et al. state:

"*graph2vec's embeddings could be used along with general purpose clustering algorithms such as K-means and relational clustering algorithms such as Affinity Propagation (AP) [14] to achieve this.*" [NCV+17, p. 5]

6.1.7 Include bidirectional edges into slices

An idea that might seem counterintuitive at first, since it violates the flow of the executed bytecode, would be to introduce bidirectional edges between nodes within slices instead of unidirectional ones. We suspect that this could be a way to put more weight onto instructions that occur later in a slice.

6.2 Limitations

The conducted experiments show that our implementation of [HHY+21] can indeed work as intended for both *1-to-1 matching* and *N-to-M matching*. Both applications, however, come with limitations that need further elaboration.

The limitations of 1-to-1 slice matching become obvious when looking at our conclusions from section 5.1. Firstly, it is a rather difficult undertaking to compile a dataset of vulnerable contracts where each contract contains slices that perfectly embody a vulnerability's logic. A too liberal selection process when marking vulnerable slices will result in many **false positives**, as we have found. Being too strict, however, will come with the drawback that either little vulnerable slices will be found, or a very large set of vulnerable contracts needs to be assembled.

Unfortunately, overcoming these obstacles does not guarantee to find vulnerabilities. We have seen that few of our matches actually passed a simple sanity check of verifying whether the instruction defining the vulnerability is present or not (Figure 5.1), even if the matched slices were subjectively similar otherwise.

We also recognize limitations of N-to-M matching. Foremost, we point out that in contrast to our experiment with wallet contracts (subsection 4.4.2), results of the experiments using standard contracts (subsection 4.4.1) were mediocre at best. We name two possible reasons for the discrepancy between these two datasets. Firstly, contracts that produce only a small number of slices (e.g. less than 50) are difficult to match with each other, especially if the extracted slices are short and of generic nature, rather than characteristic. This claim is supported by Figures 4.13 and 4.14, where we have shown that contracts with a larger number of slices exhibit a stronger correlation with the Jaccard index over function signatures.

The second reason is that the wallet dataset consists of more homogeneous contracts compared to randomly sampled ones. This, by itself, is a rather strong restriction as it suggests that our method can only be applied to contracts which are already known to be similar.

Another limitation is that the verification of our results is a difficult undertaking, especially if no source code of the matched contracts is available. However, even if source code is obtainable, it remains a difficult task to verify alleged similarity as smart contracts may span several hundreds of lines and sometimes consist of dozens of subcontracts, making manual investigation tedious and unreliable. Using the function signatures of contracts as a guide, as we have often done, sometimes is not viable, either, as we have shown with SafeMath contracts (see Table 4.11 and Figure 4.3a) which usually do not expose their internal functions.

Lastly, our method is also constrained by technical limitations. Even though we managed to reduce the runtime of our last stage by dismissing the optimal solution in favor of a heuristic one, the overall process can still take a long time to complete. While the changes we made to graph2vec comprise functionalities to stream data from disk instead of reading once and storing in memory, doing so will result in an explosion of runtime. To provide context, embedding the **full** dataset was only possible using the streaming based approach and took more than 50 hours, while using the memory based approach on the 10 % dataset typically took half an hour at roughly 70 % memory consumption.

Another technical limitation is that contracts cannot be added to corpora *after* training, meaning that all contracts have to be trained at the same time. During our research, this resulted in numerous time- and data-consuming reruns.

6.3 Closing Thoughts

This work pursued two goals. One was to implement the method of [HHY$^+$21], who tried to automatically detect vulnerabilities between smart contracts. By creating CFGs of smart contracts, extracting multiple graphs (a.k.a. slices) thereof and calculating their vector representations using the graph embedding framework graph2vec, we believe that we have succeeded in implementing a process pipeline that is close to the original of [HHY$^+$21]. Lastly, we calculate the similarities between slices deemed to be vulnerable, and slices of unknown contracts with the goal of finding vulnerable ones. While the results of our *1-to-1 slice matching* did not reach the levels of [HHY+21], we discussed the reasons why we believe that this was the case. We note that due to lack of detail, many assumptions had to be made during the process of implementing the methods of [HHY+21], which also partly explains the poor results of 1-to-1 matching. However, we also believe that we have gathered enough evidence to claim that the methods employed by Huang et al. [HHY+21] are indeed useful for the intended purpose. In section 5.1, we discuss that our method *can* perform according to the authors' description, as we showed that one type of vulnerabilities, Externally Forced Fail (EFF), was repeatedly detected by *our* implementation of the method. We also highlight the weak points of our method in this regard, and provide suggestions that we believe would lead to improved results overall. During our implementation process, we extended functionalities of the existing tools EtherSolve (section 2.7), and graph2vec (subsection 2.8.3) in regard to scalability.

The second goal of this work was to explore means of relating not only individual (vulnerable) slices to each other, but also to apply the same method of [HHY+21] in a way that allows us to detect similarities between entire contracts. By implementing an optimal matching method first, and later on sacrificing optimality for a heuristic but much more performant method, we were partly able to show that there is a moderate correlation between our similarity score and the Jaccard index over function signatures. We successfully implemented the heuristic method in such a way that it exploits the enormous parallelization capabilities of the GPU in comparison to the CPU using the TensorFlow framework. Over the span of multiple experiments, we were able to show that medium correlation between our similarity score and the Jaccard index over function signatures exists, and believe that this is an indicator *in favor* of our method. Further, we were able to use our method to prove previously formulated hypotheses about other features of contract matches (subsection 5.2.3) that correlate with our similarity metric.

In an attempt to reduce the amount of data and thus, the overall runtime, we built our datasets by regarding not just contracts with *unique* bytecode, but made use of a technique called *skeletizing*[gsa22, dAS19b], which allowed us to remove functionally identical contracts from our datasets.

Another approach we followed while aiming to reduce runtime is the use of a static stack configuration. Instead of calculating an accurate stack by simulating execution of a CFG top down, we only start our simulated execution with the same instruction that represents the slicing criterion. We believe to have shown that this is a *cheap* way to reduce runtime, though the effects of the number of elements in the static stack still need to be investigated.

We recognize that our method is still limited in many regards, but we believe that it can prove useful, especially when combined with other methods. For example, we have shown that two wallets, that were previously classified as mismatch in the wallets experiment, do in fact share almost identical semantics with each other (subsection 5.2.2). A combination of our method with ones that classify contracts based on function signatures *could* help to achieve a more fine-grained distinction between contract pairs. In our opinion, the best proof supporting this claim, and showing that our method can work as intended, are the results of the wallet experiments.

However, we also realize that the method is not yet mature enough to fully harness its potential. For the time being, we recommend to consider the method presented in this work as a proof of concept (PoC), rather than a reliable means of detecting semantic similarities between contracts, and urge for the implementation of the suggestions in section 6.1, before conducting further experiments. We believe that the original application, searching vulnerabilities using 1-to-1 matching, would benefit substantially from these suggestions as well.

List of Figures

List of Tables

Glossary

doc2vec A variation of word2vec, used to create embeddings of entire documents, rather than individual words. Useful for tasks where finding similarity of words is not sufficient. 15

ERC From https://eips.ethereum.org/:
"*Application-level standards and conventions, including contract standards such as token standards (ERC-20), name registries (ERC-137), URI schemes (ERC-681), library/package formats (EIP190), and wallet formats (EIP-85).*" 3

EtherScan A service that provides insightful data regarding the Ethereum blockchain, including the ability for developers to upload the source code of their contract, and to verify it. 70

GitHub A service that hosts public and private git repositories. 3, 4, 11, 45, 47

graph2vec A neural network for creating graph embeddings. 3, 5, 11, 15, 29, 36, 79, 90–93

OpenZeppelin A crypto and cybersecurity technology services company that maintains standard interfaces and implementations of contracts on their public GitHub repository. 45, 47, 82

SafeMath A library used for arithmetic operations to prevent integer under- and overflow bugs in smart contracts. 71, 81, 92

scipy Python-based math framework containing reference implementations of algorithms for common problems in computer science. 32

SMT solving A technique commonly used for symbolic execution of program code. 4

TensorFlow A machine learning framework for Python that is capable of running code on the GPU. 33, 34, 89, 93

word2vec A neural network by Mikolov et al. for creating semantic-preserving vector representations of words, so-called *embeddings*. 11, 15, 35

Acronyms

BID Block-info Dependency 70

CFG Control Flow Graph 3–5, 11, 17, 19, 20, 22–24, 27, 28, 35, 36, 39, 46, 52, 70, 71, 90, 92, 93, 95

DeFi Decentralized Finance 6

DoS Denial of Service 7, 9, 70, 77

EFF Externally Forced Fail 40, 70, 73–77, 87, 93, 95, 97

ERC Ethereum Request for Comment 3, 8, 10, 45, 46, 81, 82, *Glossary:* ERC

EVM Ethereum Virtual Machine 5–10, 20

JSON JavaScript Object Notation 18, 20, 36

ML Machine Learning 11

NFT Non-fungible Token 6

NN Neural Network 5, 11–13

OOP Object-oriented programming 7

PC Program counter 9

PoC proof of concept 78, 94

ReLU Rectified Linear Unit 12

solc Solidity Compiler 1, 5, 7, 17, 38–40, 45, 86

TOD Transaction Order Dependency 71

EVM Opcodes

ADD Pops two values of the stack and pushes their sum back onto it. Unsafe against overflows. 26

AND Bitwise `AND` operation. 26

BLOCKHASH Queries the hash value of one of the 256 most recent blocks and pushes it onto the stack. 9, 20, 80

CALLDATACOPY Copies the data that was provided alongside a message into the *memory* section. 71

CALLDATALOAD Reads a (u)int256 from the data provided alongside a message and pushes it onto the stack. 9, 20, 21, 26

CALLDATASIZE Queries the message data length in bytes and pushes it onto the stack. 21

CALLER Queries the address of the message caller and pushes it onto the stack. 9

CALLVALUE Queries the amount of *wei* that was provided alongside a message. 9

CALL Calls a method in another contract. Result of execution is pushed onto the stack. 10

CREATE2 Allows to create contracts at a precomputed address. 10

CREATE Creates a new contract at an address by copying the provided code to the address's code section. 10

DELEGATECALL Calls another contract which uses the storage of the calling contract as context. Useful for proxy contracts, or for updating implementation after the deployment. 78, 79

DUP Opcodes of the **DUP** family. Duplicates the N^{th} of the stack and places it on top. 10

EQ Checks for equality of the two topmost elements in the stack. Result is pushed back onto the stack. 26

GT *Greater-than* comparison between the two topmost elements in the stack. Result is pushed back onto the stack. 26

INVALID Placeholder mnemonic for all unmapped instructions 11, 25

JUMPDEST Denotes the beginning target of a conditional jump. 11

JUMPI Performs a conditional jump to a position in the code which is read from the stack. 11, 25, 26, 28

JUMP Performs a jump to a position in the code which is read from the stack. 11

LT *Less-than* comparison between the two topmost elements in the stack. Result is pushed back onto the stack. 26

MUL Multiplies the two topmost elements of the stack and pushes the result back onto it. Unsafe against overflows. 26

NUMBER Queries the current blocks number and pushes it onto the stack. 9

ORIGIN Queries the origin from the transaction data and pushes it on top of the stack. 72

OR Bitwise `OR` operation. 26

PUSH Opcodes of the **PUSH** family. Pushes a value onto the stack. 10, 26

RETURN Returns from current contract call. Return value is specified by the two topmost elements in the stack and are read off the memory. 11

REVERT Stops the execution and reverts all state changes. 11

SELFDESTRUCT The contract empties its storage and code sections and transfers remaining funds to a specified address. 10, 11, 73

SGT Signed *greater-than* comparison between the two topmost elements in the stack. Result is pushed back onto the stack. 26

SLT Signed *less-than* comparison between the two topmost elements in the stack. Result is pushed back onto the stack. 26

STOP Halts the execution of the contract. 11

SWAP Opcodes of the **SWAP** family. Swaps the topmost value of the stack with the N^{th}. 10

XOR Bitwise exclusive `OR` operation. 26

Bibliography

[10.63] The quadratic assignment problem. *Manage. Sci.*, 9(4):586–599, jul 1963.

[AYCO21] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts. *BSCI 2021 - Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, co-located with ASIA CCS 2021*, pages 47–59, 2021.

[Ber18] Mueller Bernhard. Smashing ethereum smart contracts for fun and real profit. https://github.com/muellerberndt/smashing-smart-contracts/blob/0663ad015b0a6ce08053d48731cdee1e7bc4e726/smashing-smart-contracts-1of1.pdf, 2018. [Online; accessed 2022-04-15].

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Etherum*, (January):1–36, 2014.

[CCCP21] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. *IEEE International Conference on Program Comprehension*, 2021-May:127–137, 2021.

[Coh88] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 1988.

[Con21] ConsenSys. Mythril. https://github.com/ConsenSys/mythril/blob/dfaa9382ecbf6387e12e121c6368a95a6f3342ed/README.md, 2021. [Online; accessed 2022-04-15].

[Cro16] David F. Crouse. On implementing 2D rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1679–1696, 2016.

[dAS19a] Monika di Angelo and Gernot Salzer. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, Piscataway, NJ, USA, apr 2019. IEEE.

[dAS19b] Monika di Angelo and Gernot Salzer. Mayflies, breeders, and busy bees in ethereum: Smart contracts over time. *BCC 2019 - Proceedings of the 3rd ACM Workshop on Blockchains, Cryptocurrencies and Contracts, co-located with AsiaCCS 2019*, pages 1–10, 2019.

[dAS20a] Monika di Angelo and Gernot Salzer. Assessing the similarity of smart contracts by clustering their interfaces. *Proceedings - 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020*, pages 1910–1919, 2020.

[dAS20b] Monika di Angelo and Gernot Salzer. Characterizing Types of Smart Contracts in the Ethereum Landscape. In *4th Workshop on Trusted Smart Contracts, Financial Cryptography*. Springer, 2020.

[dAS20c] Monika di Angelo and Gernot Salzer. Tokens, Types, and Standards: Identification and Utilization in Ethereum. In *Int. Conf. Decentralized Applications and Infrastructures (DAPPS)*, pages 1–10. IEEE, 2020.

[dAS20d] Monika di Angelo and Gernot Salzer. Wallet Contracts on Ethereum – Identification, Types, Usage, and Profiles. 2020.

[DP14] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)*, 61(1):1–23, 2014.

[Eth22a] Ethereum. Application binary interface specification — solidity 0.4.21 documentation. https://docs.soliditylang.org/en/v0.4.21/abi-spec.html#function-selector, 2022. [Online; accessed 2022-03-24].

[Eth22b] Ethereum. Language influences — solidity 0.8.14 documentation. https://docs.soliditylang.org/en/latest/language-influences.html, 2022. [Online; accessed 2022-03-19].

[FFB19] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 93–112, Cham, 2019. Springer International Publishing.

[FTSS18] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 664–676, New York, NY, USA, 2018. Association for Computing Machinery.

[GK11] Erkam Guresen and Gulgun Kayakutlu. Definition of artificial neural networks with comparison to other networks. *Procedia Computer Science*, 3:426–433, 2011. World Conference on Information Technology.

[gsa22] gsalzer. Skeletizing bytecode. https://github.com/gsalzer/ethutils/tree/main/doc/skeleton, 2022. [Online; accessed 2022-03-26].

[HHY+21] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Transactions on Information Forensics and Security*, 16:2144–2156, 2021.

[HSR+18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. *Proceedings - IEEE Computer Security Foundations Symposium*, 2018-July:204–217, 2018.

[HYL21] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. Survey on blockchain based smart contracts: Applications, opportunities and challenges. *Journal of Network and Computer Applications*, 177(November 2020):102857, 2021.

[Koo92] Philip Koopman. A preliminary exploration of optimized stack code generation. 1992.

[Kow72] Charles J. Kowalski. On the effects of non-normality on the distribution of the sample product-moment correlation coefficient. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 21(1):1–12, 1972.

[Kuh55] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics*, 52(1):7–21, 1955.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, New York, NY, USA, oct 2016. ACM.

[LYJ+19] Han Liu, Zhiqiang Yang, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Enabling Clone Detection for Ethereum via Smart Contract Birthmarks. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 105–115. IEEE Press, 2019.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, pages 1–12, 2013.

[MtSD07] G Ayorkor Mills-tettey, Anthony Stentz, and M Bernardine Dias. The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs. *Naval Research Logistics Quarterly*, (July):83–87, 2007.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, 2008.

[NCV⁺17] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning Distributed Representations of Graphs. *28th Modern Artificial Intelligence and Cognitive Science Conference, MAICS 2017*, pages 189–190, jul 2017.

[Pic22] Lukas Pichler. Collaborative Inference for Edge Intelligence - Impacts on Performance and Privacy of Partition Points, 2022.

[QTN21] Ilham Qasse, Manar Abu Talib, and Qassim Nasir. *Toward Inter-Blockchain Communication Between Hyperledger Fabric Platforms*, pages 251–272. Springer International Publishing, Cham, 2021.

[RBC⁺98] Martin Röscheisen, Michelle Baldonado, Kevin Chang, Luis Gravano, Steven Ketchpel, and Andreas Paepcke. The Stanford InfoBus and its service layers: Augmenting the internet with higher-level information management protocols. pages 213–230, 1998.

[RdAS22] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. Review of automated vulnerability analysis of smart contracts on ethereum. *Frontiers in Blockchain*, 5, 2022.

[TDC⁺18] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. *arXiv*, jun 2018.

[TPF⁺09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[Woo14] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014.

[WS21] Akhilesh A. Waoo and Brijesh K. Soni. Performance analysis of sigmoid and relu activation functions in deep neural network. In Amit Sheth, Amit Sinhal, Abhinav Shrivastava, and Amit Kumar Pandey, editors, *Intelligent Systems*, pages 39–52, Singapore, 2021. Springer Singapore.

[XLF⁺17] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 363–376, 2017.