UCLouvain

epl

École polytechnique de Louvain

Improving function signature in malware analysis using neural networks

Author: **Alix Temmerman**
Supervisor: **Axel Legay**
Readers: **Tom Barbette, Ramin Sadre, Serena Lucca**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

Binary code similarity detection is a critical task in numerous security applications such as malware analysis, bug search, and software theft detection. This thesis explores the applications of the SAFE (Self-Attentive Function Embeddings) tool to enhance the SEMA-Toolchain by implementing a novel method for detecting similar functions in binary code. SAFE leverages a self-attentive neural network to generate function signatures, which are then used to identify similar functions across different binaries.

The proposed method is evaluated in samples of several malware families such as Warzone and Satan. We demonstrate that our approach is capable of detecting similar functions in binaries more effectively and accurately than the current method used in the SEMA-Toolchain. Furthermore, we show that our method can detect common functions in different malware families. Finally, we demonstrate that our method can be used to improve the SEMA-Toolchain and is able to detect similar functions in binaries compiled from the same source code but with different compilers and compiler optimizations.

Finally, we discuss the limitations of our work and propose future research directions to further enhance the performance of binary code similarity detection.

Acknowledgements

First, I would like to thank my supervisor Prof. Axel Legay for his guidance and encouragement throughout this year. I would also like to express my deepest gratitude to Serena Lucca for her consistent support, quick responses and guidance throughout this project. I would like to thank Samy Bettaieb and Charles-Henry Bertrand Van Ouytsel for their help and support all year long. Finally, I would like to thank Christophe Crochet for his help and guidance during the writing of this thesis.

Furthermore, I would like to thank the readers and members of the jury, Prof. Axel Legay, Prof. Ramin Sadre, Prof. Tom Barbette and Serena Lucca for taking the time to review this work.

Finally, on a more personal note, I would like to thank my family for their support and encouragement throughout my studies and more specifically during this challenging year. I would also like to thank my friends for their support and for always being there for me.

Contents

1 Introduction **1**

2 Background Knowledge And Literature Review **4**
2.1 Symbolic Execution 4
2.1.1 SEMA-Toolchain 5
2.2 Binary Similarity Problem 9
2.2.1 Static Analysis 9
2.2.2 Dynamic Analysis 12
2.3 SAFE - Self-Attentive Function Embeddings 13
2.3.1 Overview 14
2.3.2 i2v Model 14
2.3.3 Self-Attentive Neural Networks 15
2.3.4 Models Training 17
2.4 Other Tools 18
2.4.1 Ghidra 18
2.4.2 Radare2 20

3 Contribution **21**
3.1 Finding functions in binaries 21
3.1.1 Signature creation 22
3.1.2 Signature matching 22
3.2 Automatic threshold selection 23
3.3 Finding common functions in malware families 24
3.4 Integration in SEMA-Toolchain 26
3.5 Challenges 27

4 Experimental Results And Analysis **28**
4.1 Function Identification In Binaries 28
4.1.1 Warzone 29
4.1.2 Warzone Compiled With Visual Studio 31
4.1.3 Satan 32

4.1.4 Conclusion 32
4.2 Threshold Selection 33
4.3 Finding Common Functions In Malware Families 34
4.4 Use Case1: SEMA-Toolchain 37
4.5 Use Case2: Finding Functions In Binaries Compiled With Different Compilers 40

5 Limitations And Future Work **43**

6 Conclusion **45**

Chapter 1

Introduction

In recent years, there has been a noticeable increase in the quantity and complexity of malicious software, commonly known as malware. Statistics indicate that the monetary damage caused by cybercrime in the United States has increased by 21% in 2023 compared to 2022 [1]. In 2023, the total damage caused by malware reached a historical peak estimated at 12.5 billion dollars.

During the early days of malware, in the 1980s, malware was relatively simple and easy to detect. It was typically transferred between computers via floppy disks [22] and was initially used as a form of prank, but soon evolved into more malicious software. In the 1990s, malware authors began using obfuscation techniques to conceal their code. This rendered the initial generation of antivirus software ineffective as it was not designed to detect obfuscated code. Nowadays, there exists a wide range of malware including viruses, worms, trojans, ransomware, etc. Malware authors employ an array of sophisticated obfuscation techniques to conceal their code such as control flow flattening, code reordering and dead code insertion [8].

Fortunately, cybersecurity researchers have developed a wide range of tools to detect malware and protect users and companies. Some of these tools employ static analysis, while others utilize dynamic analysis. Static analysis consists of analyzing the code without executing it, focusing on aspects such as the code structure, the control flow graph and the system calls. In contrast, dynamic analysis involves executing the code and analyzing its behavior during execution. Both approaches have their respective advantages and drawbacks. Static analysis is more susceptible to obfuscation techniques whereas dynamic analysis can be escaped by malware. If a malware detects that it is being analyzed, for instance, if it detects that it is being run in a sandbox, it can modify its behavior in order to avoid detection [17].

This thesis will examine another approach to malware detection: symbolic

execution. Symbolic execution is a static analysis technique that involves executing the code with symbolic values instead of concrete values. A symbolic execution engine executes the code and generates constraints on the symbolic values. Those constraints can then be used to assess the feasibility of a given path. This approach allows for the analysis of code without executing it.

The SEMA-Toolchain [10] is a framework that applies symbolic execution to malware analysis. It analyses binaries to first generate a system dependency call graph and then uses machine learning to classify the binary as malicious or benign. The Toolchain is able to detect malware with high accuracy [11]. However, some functions are very complex and thus slow to execute through the symbolic execution engine. Consequently, they are replaced by a procedure that summarizes the behavior of the function. Currently, SEMA employs pattern matching to identify known functions in binaries [18]. Each pattern must be manually extracted from an analyzed binary. This process is both time-consuming and not scalable. This approach also results in a high number of false negatives, as malware authors frequently employ obfuscation techniques to conceal their code. The obfuscation techniques employed by malware authors result in alterations to the byte sequence of the function, thereby affecting the signature of the function.

Binary code similarity detection is the process of determining whether two or more pieces of binary code are similar without examining the source code. This problem is of significant importance in numerous security applications, including bug search, malware clustering, detection and lineage, patch generation and analysis, and software theft detection [15]. A number of methods have been proposed to address this issue, including fuzzy hashing and control flow graph matching, among others. In this thesis, we will utilize SAFE, a tool that employs a self-attentive neural network to generate function signatures that can be utilized to identify similar functions within binary code.

In this work, we enhance the SEMA-Toolchain by applying binary code similarity detection. We propose a method to detect similar functions in binaries using SAFE. The efficacy of our method was evaluated on a number of malware samples, demonstrating its capacity to identify similar functions within binaries. We show that our method not only reduces the number of signatures required to detect functions in binary but also identifies a greater number of functions than the current method used in the SEMA-Toolchain. Furthermore, we utilize binary code similarity detection to identify shared functions across distinct malware families. Finally, we demonstrate that our method can be employed to enhance the SEMA-Toolchain.

The remainder of this thesis is organized as follows. Chapter 2 provides the background information necessary to comprehend the subsequent sections of the thesis. The following sections will introduce symbolic execution and the SEMA-Toolchain. We then address the binary similarity problem and the various forms of analysis. Finally, we present SAFE, the tool that we will use to enhance the functionality of function signatures within the SEMA-toolchain, and other useful tools for malware analysis such as Ghidra and radare2. Chapter 3 presents the implementation of the binary code similarity detection in the SEMA-Toolchain. Chapter 4 presents the experimental results on several malware samples. At the end of this thesis, we will discuss the limitations of this work and discuss future work in Chapter 5. Finally, this thesis will conclude with a discussion of its contributions and a summary of its findings in Chapter 6.

Chapter 2

Background Knowledge And Literature Review

This chapter provides a review of the state of the art in binary code similarity detection, as well as all the background knowledge needed for the understanding of the thesis. In section 2.1, we will introduce the concept of symbolic execution and discuss the challenges associated with it. Subsequently, we present the SEMA-Toolchain in section 2.1.1 which is the tool used in our first use case. In section 2.2, we present the binary similarity problem and the different approaches, static and dynamic, to binary similarity detection. We then present SAFE in section 2.3 which is the main tool used in this thesis. Finally, in section 2.4, we present other tools that we used to analyze binaries throughout this thesis. These include Ghidra and radare2.

2.1 Symbolic Execution

Symbolic execution is a technique used to analyze the binary code without executing it dynamically. The technique was first used in the 1970s to verify whether a program respected specific properties [9]. For instance, it could be used to verify that no division by zero could occur in a program. While concrete execution is constrained to exploring a single path that a program takes with the given input, symbolic execution is capable of exploring multiple paths that a program could take with different inputs. It explores all the potential execution paths by representing the values as symbolic values on which constraints are applied. A model checker is then used to verify whether the constraints are satisfiable.

Symbolic execution presents a number of challenges. The first challenge is

the state space explosion problem. Upon encountering a condition, the symbolic execution engine generates two new branches: one where the condition is respected and one where it is not. This results in an exponential growth in the number of paths to be explored, which in turn limits the number of paths that can be searched. Since we can only explore a finite number of paths, it is necessary to select which paths will be searched and which will not. This can be done by using an exploration strategy such as the Depth-First Search (DFS) strategy or the Breadth-First Search (BFS) strategy, among others. Another challenge is the handling of system calls. System calls are used to interact with the operating system. They are used to perform tasks such as reading and writing to files, creating processes, etc. Such calls must be handled by the symbolic execution engine.

Despite its efficacy, symbolic execution is a particularly time-consuming technique. This is due to the state space explosion problem. The number of paths that need to be explored grows exponentially with the number of conditions in the code. It is thus impractical for large programs. In the next section, we will present the SEMA-Toolchain, a tool that uses symbolic execution to analyze, detect and classify malware and the solution it provides to the state space explosion problem and the handling of system calls.

2.1.1 SEMA-Toolchain

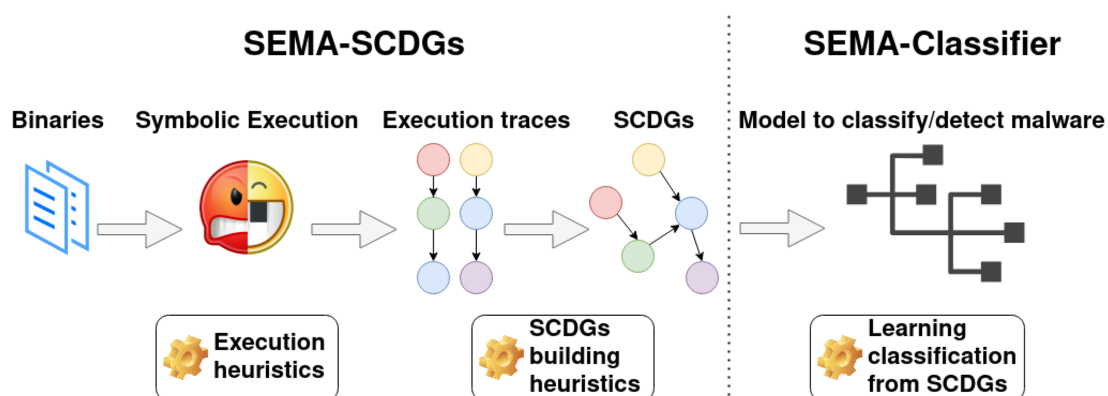

Figure 2.1: SEMA-Toolchain workflow [13]

The SEMA-Toolchain [11] is an open-source framework that is used for the analysis, detection and classification of malware. SEMA stands for Symbolic Execution for Malware Analysis. The toolchain workflow is illustrated in figure 2.1. It can be divided into three main components: the symbolic execution engine, the system

call dependency graph (SCDG) building tool and the classifier. In this section, we will present the three components of the SEMA-Toolchain as well as highlight the features that are used in this thesis.

Symbolic Execution Engine

The symbolic execution engine is the first component of the SEMA-Toolchain. It is used to analyze the binary code and to extract the system call dependency graph (SCDG). In the SEMA-Toolchain, the symbolic execution engine is based on the Angr [30] symbolic execution engine. Angr is a powerful open-source symbolic execution engine that can be used to analyze binary code. It is written in Python and has a large community on GitHub.

The first challenge that must be addressed is the handling of system calls and library calls. The solution proposed by Angr is to use functions that summarize the behavior of the function they replace. These functions are called SimProcedures. The process of replacing the functions with SimProcedures is illustrated in figure 2.2. To identify the functions that need to be replaced by SimProcedures, Angr uses hooks. The hooks are placed at the beginning of the execution and later, the functions that are hooked are replaced by the SimProcedures. This allows the symbolic execution engine to simulate the behavior of the system calls and the library calls.

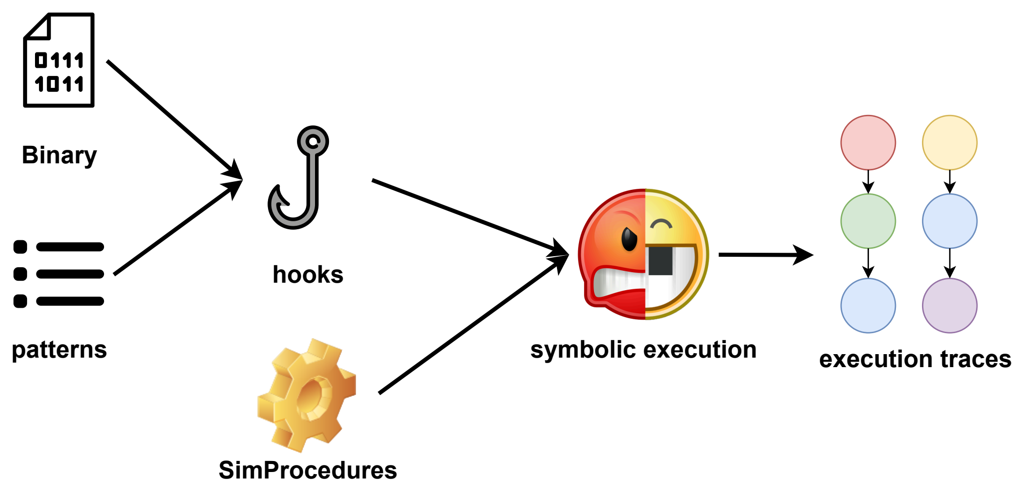

Figure 2.2: Hooking process in the SEMA-Toolchain

The second challenge that must be addressed is the state-space explosion problem. The number of paths that must be considered grows exponentially with the number of conditions present in the code. To address this issue, the SEMA-Toolchain uses the principle of SimProcedures. Functions that have a complex behavior and that slow down the symbolic execution by creating a lot of paths are replaced by SimProcedures. This enables the symbolic execution engine to explore a greater number of paths and to delve more deeply into the exploration process.

Currently, the functions that need to be hooked are identified through pattern matching. This is done by searching for specific patterns in the binary code. This is a slow process as each of the patterns needs to be manually added to a list of patterns. However, the gain in speed of the symbolic execution engine is significant enough to justify the time spent on adding the patterns. Another challenge of this approach is that the patterns may be modified by obfuscation techniques. Several patterns are needed to identify each function and even with many patterns, some functions can be missed. A change as small as an address change is enough to miss a function. In this thesis, we aim to improve the function identification in the SEMA-Toolchain by reducing the number of signatures needed to find a function and by improving the accuracy of the function identification.

During the symbolic execution, the system calls encountered are printed, along with the different steps of the symbolic execution. A summary of the execution is also provided. It contains a comprehensive list of the system calls that were encountered, the loaded libraries, the execution time, and other pertinent information. This information can be used as a metric to measure how far in the exploration of the binary the engine went.

SCDG Building Tool

The SCDG building tool is the second component of the SEMA-Toolchain. The system call dependency graph (SCDG) is created using the paths explored by the symbolic execution engine, as depicted in figure 2.3. The first step is to extract the system calls from the paths, along with their respective addresses and the arguments of the calls. They are then used to create the SCDG. The system calls are represented as nodes in the graph and the edges represent the information flow between the system calls.

In order to reduce the size of the resulting SCDG, the SEMA-toolchain uses a merging strategy that combines system calls with identical API names and identical addresses, provided that they occur within the same symbolic path and differ only

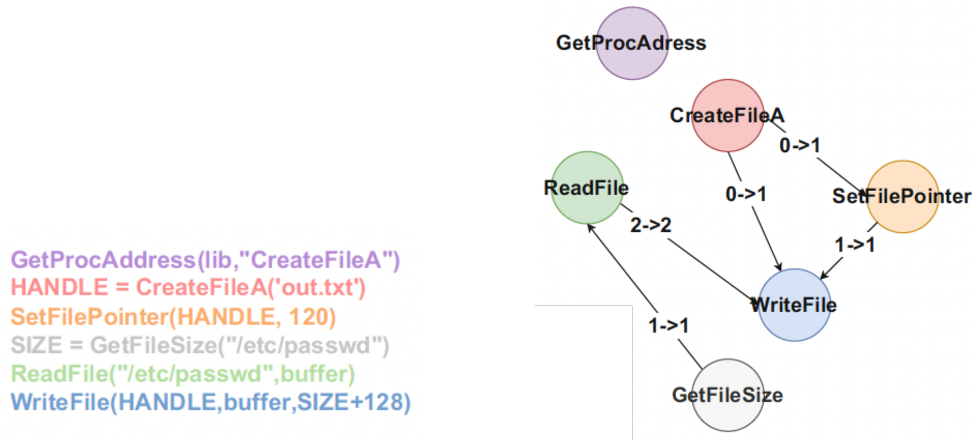

Figure 2.3: Example of a SCDG from [11]

in their arguments. This presents several advantages. Firstly, this process results in a reduction in the size of the SCDG. Secondly, it reduces the impact of redundant system calls during the learning phase. This strategy is called SCDG-Strategy 3. The Sema-Toolchain also uses two other strategies to merge system calls, the SCDG-Strategy 4 that merges successive executions from different symbolic paths and SCDG-Strategy 5 that merges several symbolic paths into one SCDG by keeping the disjoint union of the system calls.

Classifier

The last component of the SEMA-Toolchain is the classifier. It is used to classify the malware. It takes as input the system call dependency graph (SCDG). The toolchain implements two distinct classifiers. The first classifier implements the gspan algorithm [26]. This approach computes a signature for each malware family by computing the largest subgraph between the SCDGs of the malware family. The second classifier is based on the graph kernel approach presented in the work of Puodzius et al. [25]. The algorithm generates a Gram matrix which represents the similarity between the SCDGs and can then be used by a support vector machine to classify the malware.

Features Used In This Thesis

In this thesis, we aim to improve function identification in the SEMA-Toolchain to enhance the hooking mechanism. We use the system calls as well as the summary

that are printed by the symbolic execution engine to measure how many different symbolic paths are explored. This allows us to measure whether we improved the function identification. The number of signatures required to identify the functions is used as a metric to assess the performance of the tool.

2.2 Binary Similarity Problem

The binary similarity problem refers to the task of detecting whether two pieces of code are similar at the binary level without access to the source code. The first challenge in this problem is to define what it means for two functions to be similar. Similarity can be defined in many ways. Some approaches consider that two functions are similar if they have a similar syntactic structure, other approaches consider that two functions are similar if they have a similar semantic. Some approaches consider that two functions are similar if they have a similar control flow graph or similar features. In this thesis, we use SAFE, a tool that uses a semantic approach to identify binary code similarity and considers that two binary codes are similar if they have been compiled from the same source code.

There are two main challenges in the binary similarity problem. The first challenge is that the same piece of code can give very different binaries after compilation. Indeed, the same source code compiled for different architectures, with different compilers, different optimization levels, different flags, etc will give different binary codes. Furthermore, obfuscation techniques can be used both on the source code and the binary code which makes the detection of similarity even more difficult. A second challenge is that a large part of the semantics of the program is lost during the compilation process. Indeed, the function names, the variable names, the comments, the structure of the code, etc. are all lost during the compilation process. This makes it difficult to understand what the code does and thus to detect whether two functions have the same behavior or features.

In this section, we will present the different approaches to binary similarity detection and the challenges that arise in this problem.

2.2.1 Static Analysis

Static analysis is the most common approach to binary code similarity detection. It consists of analyzing the binary code without executing it. Several approaches have been proposed to detect binary code similarity using static analysis. Some

approaches use syntactic and structural information to create the function's signature, whereas others use program simulation to determine whether two functions are similar. Finally, some approaches use machine learning to create embeddings. In this section, we will detail each of those approaches and present some tools that implement them.

Syntactic And Structural Analysis

The first approach consists of extracting syntactic and structural information from the binary code and then comparing this information to detect similarities. This approach regroups several techniques, including the use of opcode sequences to detect binary code similarity as proposed by Idea [27]. They compute the weighted term frequency for every opcode sequence of fixed length and use it as a signature. The signature can then be compared to detect similarities with other binary codes.

Another technique consists of extracting features from the disassembled binary code and then comparing these features to detect similarities. The features can be extracted from the control flow graph, the call graph or directly from the binary code. For instance, binshape [29] extracts instruction-level features, including the number of instructions, the number of arguments, the return type, etc. It also extracts graph features such as the number of nodes, the number of edges, the number of connected components, etc. Finally, it extracts statistical features such as the skewness and the kurtosis measures and converts them to scores. A feature selection process is then run to select the most relevant features. The similarity between two functions is then computed using these features. The signature of a function is composed of heterogeneous features, including instruction-level features, graph features, and statistical features to make it more robust to some code transformations. However, it is not sufficient to counter advanced obfuscation techniques such as control flow flattening. Furthermore, packed, encrypted and obfuscated binary codes are not supported.

Syntactic and structural analysis are highly efficient, as the features extracted from the binary code are relatively inexpensive to compute. However, it is not robust to obfuscation techniques, as the control flow graph and the opcode sequences can be modified by obfuscation techniques such as control flow flattening and dead code insertion. It is thus not suitable for detecting similarity in malware which are often obfuscated.

Program Simulation

A second approach consists of simulating the program to identify similarities. This approach is used by Luo et al. [19] with the tool CoP. Their method consists of first extracting the control flow graph. A set of symbolic formulas that represent the input-output relations of basic blocks is then computed with symbolic execution. To identify similarities, the symbolic formulas are compared via theorem proving. The longest common subsequence of semantically equivalent basic blocks is then computed to detect similarities. This approach is more robust to obfuscation techniques as it uses the semantics of the program to detect similarities. However, it is not without its shortcomings. First, it is more expensive as it requires symbolic execution and theorem proving. This approach is also sensitive to encryption and packing which are common obfuscation techniques in malware.

Another way of using program simulation to detect similarities is described by Schrittwieser et al. [28]. They base their work on the assumption that if the input-output pair of two functions is similar, then the functions are similar. SIMID, the tool proposed by Schrittwieser et al. uses symbolic execution to simulate the execution of the functions with known inputs and then compares the outputs to detect similarities. The main challenge of this approach is to find the right way to pass the inputs to the functions as their structure must match the structure of the function. This approach works across different architectures and is robust to obfuscation techniques as it does not rely on the control flow graph or the opcode sequences. The main limitation of this work is its sensitivity to modifications of the function prototype. For instance, adding arguments to the function will change the input-output relation, thereby affecting the degree of similarity.

Machine Learning

The last approach consists of using machine learning to identify binary code similarity. Machine learning is used to create embeddings of the binary code and then compare these embeddings to detect similarities. Some approaches require the extraction of a large number of features from the binary code before creating the embeddings. For example, James Patrick-Evans et al. [24] first extract a variety of code features and local and global context features from the binary code before using an autoencoder to create the embeddings. Xu et al. [32] on the other hand use a neural network to generate embeddings of the functions based on an attributed control flow graph of the function. The attributed control flow graph is created by extracting the control flow graph of the function and then adding a set of attributes as labels to each vertex of the graph. The attributes may include the number of

calls, the number of instructions, the number of offspring, etc. The neural network then generates embeddings of the functions based on the attributed control flow graph.

Both of these approaches require manually selected features to create the embeddings. This can be a drawback as the features may not be relevant for detecting similarity and may introduce a bias in the resulting embeddings. Furthermore, the features, such as the control flow graph, may be expensive to compute and might not add much information to the embeddings. Asm2Vec [14] is a tool that uses a PV-DM model, a neural network model used in natural language processing that learns vectorized representation of text paragraphs, to create embeddings of binary functions. It does not require any manual feature selection and creates the embeddings based on the opcode sequences. The embeddings are then compared to detect similarities. This approach is more efficient as it does not require any manual feature selection but it suffers from two main drawbacks. The first limitation is that it is designed for a single architecture. The second weakness is that this approach assumes that the call symbols are available in the binary code. This is not always the case as the call symbols can be stripped from the binary code.

SAFE [20] is a tool that uses a self-attentive neural network to create embeddings of the binary code. This is the approach chosen in this thesis. SAFE is designed to be architecture-agnostic and does not require any manual feature selection. The embeddings are then compared to detect similarities. We will present SAFE in more detail in the next section.

2.2.2 Dynamic Analysis

Another approach to binary code similarity detection is dynamic analysis. Dynamic analysis consists of analyzing the binary code by executing it and observing its behavior. The main advantage of dynamic analysis is that it does not rely on the binary code itself but on its behavior. This makes it more robust to obfuscation techniques. In this section, we will present two different approaches to binary code similarity detection using dynamic analysis.

The first approach is based on the same assumption as SIMID [28]. It assumes that if the input-output pair of two functions is similar, then the functions are similar. This approach is used by Calvet et al. [12] to identify similarities between functions. They first gather several traces of the target program and extract the input-output pairs of the functions. The input-output pairs are then compared to detect similarities. The main limitations of this work are that first, it requires

specific architectures and operating systems to work. Second, it is unable to detect some functions such as RSA due to the current definition and management of the loops.

The second approach is a hybrid approach that combines static and dynamic analysis. BinSim [23] is a tool that takes two executables as input and first identifies the system calls of both programs via a dynamic analysis of the two executables at the same time with the same input and environment. The system calls of the two programs are then matched thanks to a bioinformatics-inspired algorithm. Next, the arguments of the system calls are traced back to determine instructions that impact the argument's values. Finally, the weakest preconditions, which are the conditions that must be satisfied for a given instruction to be executed, are computed thanks to symbolic execution and compared with a constraint solver to detect similarities.

Shortcomings

Although dynamic analysis is more robust to obfuscation techniques, it comes with its own set of challenges. As pointed out by Schrittwieser et al. [28], dynamic analysis is sensitive to the environment in which the binary code is executed. This means that an environment with the right computing architecture and the right operating system must be available to execute the binary code. Furthermore, the environment should be isolated from the rest of the system when analyzing unknown binary codes to avoid any damage to the system in case the binary code is malicious. A second challenge is that dynamic analysis can only analyze code that is actually executed at runtime. This is a challenge as some malware can detect that the environment is a sandbox and change its behavior to avoid detection.

2.3 SAFE - Self-Attentive Function Embeddings

SAFE [20] is the tool used in this thesis to detect binary code similarity. The main idea behind SAFE is to create embeddings of the binary code and then compare these embeddings to detect similarities. The embeddings are created using a self-attentive neural network. The main advantage of SAFE is that it is architecture-agnostic and does not require any manual feature selection. In this section, we will present the architecture of SAFE and the different components of the tool.

2.3.1 Overview

SAFE is a tool used to detect binary code similarity. In this work, two binary codes are considered similar if they are compiled from the same source code. This does not make the problem trivial since the same source code can give very different binary codes after compilation. Indeed, the same source code compiled for different architectures, with different compilers, different optimization levels, different flags, etc will give different binary codes. Furthermore, obfuscation techniques can be used both on the source code and the binary code which makes the detection of similarity even more difficult.

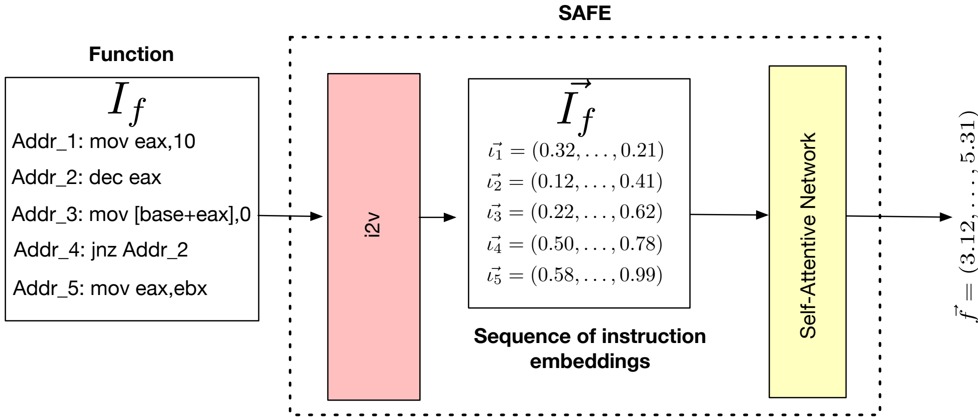

Figure 2.4: architecture of SAFE [20]

This tool works directly on the binary code and does not require the source code to be available. It also does not assume the presence of call symbols in the binary code, making it more robust to obfuscation techniques. SAFE is designed to be architecture-agnostic and does not rely on manual feature selection, reducing potential bias. The main components of SAFE are the i2v model and the self-attentive neural network, as shown in Figure 2.4. The i2v model maps assembly instructions to vectors, while the self-attentive neural network generates function embeddings. Similarities between embeddings are then used to detect similarities between binary code functions.

2.3.2 i2v Model

The first part of SAFE is the i2v model. This model takes as input a disassembled binary code and maps each instruction of the binary code to a vector. It is based on the word2vec [21] model that is extremely popular for creating embeddings of

words in natural language processing. It uses the skip-gram method which consists of using the current instruction to predict the surrounding instructions.

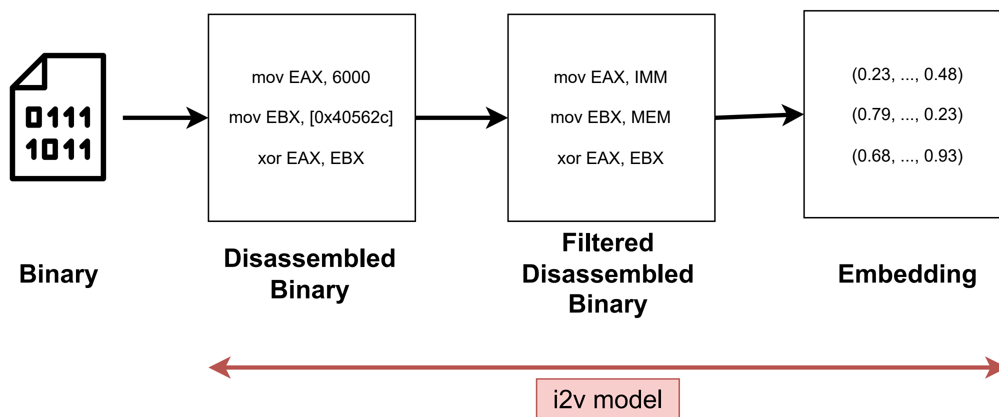

Figure 2.5: architecture of the i2v model

The architecture of the i2v model is illustrated in figure 2.5. The i2v model does not use directly the disassembled instructions but a filtered version of them. The operands of the instructions are examined and the base memory addresses are replaced with the symbol MEM and the immediate values above 5000 (this is an arbitrary threshold) are replaced with the symbol IMM. This is done to reduce the number of unique tokens and improve the quality of the embeddings. The output of the i2v model is a sequence of vectors that represent the instructions of the binary code.

2.3.3 Self-Attentive Neural Networks

The second part of SAFE is a self-attentive neural network. It takes as input the instruction embeddings generated by the i2v model and computes a summary embedding of the function. The network consists of a bi-directional recurrent neural network (RNN) that processes the instruction embeddings and computes a summary vector of the assembly vectors. Intuitively, the summary vector is a weighted sum of the assembly vectors.

The overall architecture of the self-attentive neural network is shown in the figure 2.6. The network consists of three main components: a bi-directional RNN, an attention mechanism, and a two-layer fully connected network with ReLU activation.

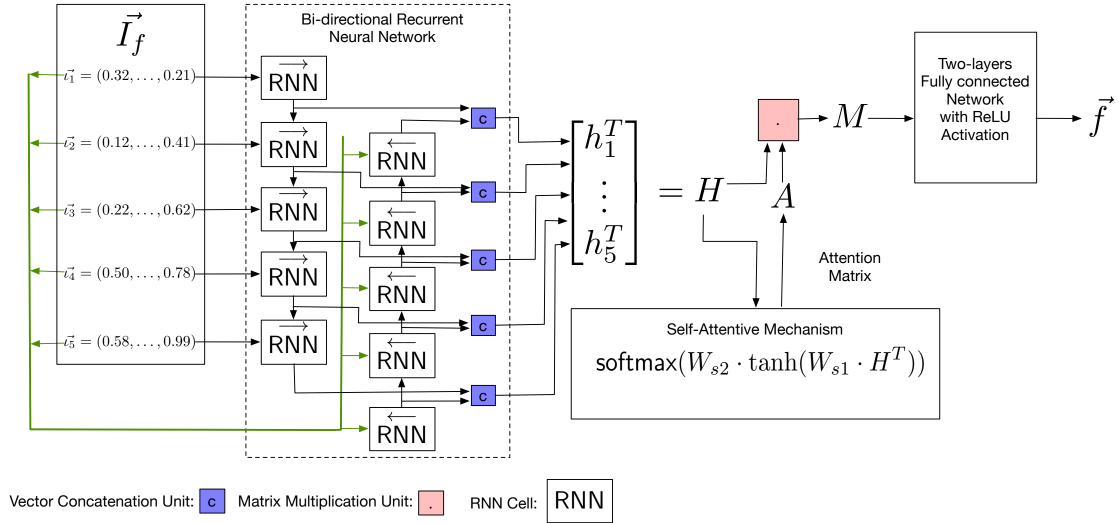

Figure 2.6: architecture of the self-attentive neural network [20]

The bi-directional RNN is the first part of the self-attentive network. It processes the instruction embeddings and computes for each embedding a summary vector of the instruction itself and its context. The summary vector is the concatenation of the forward and backward hidden states of the RNN.

The attention mechanism is the second part of the self-attentive network. It takes as input a matrix consisting of the summary vectors. The attention mechanism computes a weight for each summary vector. The weight is computed using a two-layer fully connected network. There are two parameters to the attention mechanism: the attention depth which is determined by the size of the matrix $W_{s1} = d_a \times u$ where d_a is the attention depth and u is the size of the summary vectors. The second parameter is the number of attention hops which is determined by the size of the matrix $W_{s2} = r \times d_a$ where r is the number of attention hops. The output of the attention mechanism is a matrix containing the weighted summary vectors. There are as many summary vectors as there are attention hops.

The last part of the self-attentive network is a two-layer fully connected network with ReLU activation. This network is used to compute the final embedding of the function from the matrix of the weighted summary vectors. The final embedding is then used to compare the functions and detect similarities.

2.3.4 Models Training

i2v Model

The model used for the i2v is the skip-gram model as implemented for word2vec [7]. Two skip-gram models were trained, one for the ARM instruction set and one for the x86 instruction set. The models were trained on the assembly instructions of the training set. The models were trained to produce 100-dimensional embeddings. Two datasets were created to train the two models. Both datasets were created by disassembling UNIX binaries from repositories of Debian packages with IDA Pro.

Self-Attentive Neural Network

The parameters of the self-attentive network were learned using a Siamese network represented in figure 2.7. The Siamese network takes two inputs, in this case, two functions, and computes a similarity score between them.

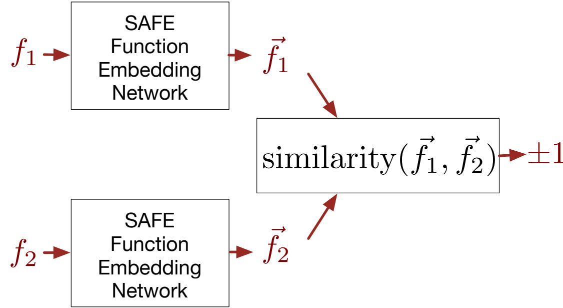

Figure 2.7: siamese network [20]

The network consists of two identical self-attentive networks with the same weights. The two functions are passed through the two networks and the embeddings of the functions are computed. The embeddings are then compared using a cosine similarity function. The network is trained by minimizing the following objective function:

$$J = \sum_{i=1}^{K} (similarity(\vec{f_1}, \vec{f_2}) - y_i)^2 + ||A \cdot A^T - I||_F \quad (2.1)$$

Where y_i is the ground truth similarity score, A is the attention matrix and I is the identity matrix.

The first term of the objective function is the loss function. It is used to minimize the difference between the predicted similarity score and the ground truth similarity score. The second term is used to ensure that the same weights are not

used for each attention hop. This is done by penalizing the attention matrix if it is close to being symmetric.

To train the network, two datasets were created, an "AMD64multiplecompilers" dataset and an "AMD64ARMOpenSSL" dataset. The "AMD64multiplecompilers" dataset was created by compiling the source code from different libraries for the AMD64 architecture with different compilers such as gcc-5.4, gcc-3.4 and clang-3.9. Different optimization levels were also used. The "AMD64ARMOpenSSL" dataset was created using two versions of the OpenSSL library that have been compiled for AMD64 and ARM using 4 different optimization levels. Each function in the datasets is then paired with two functions from the same dataset, one similar and one dissimilar and a ground truth label is assigned to each pair.

2.4 Other Tools

In this section, we will introduce two open-source tools used in binary analysis: Ghidra and radare2.

2.4.1 Ghidra

Ghidra is an open-source software developed by the NSA. It is used to analyze binary code and to reverse engineer it. This tool is very powerful and can be used to decompile and analyze the binary code. It can analyze code for many different architectures. This tool provides a number of features to help reverse engineers. The first feature is the disassembler. When a binary is first loaded in Ghidra, it is disassembled and it is displayed as a list of instructions that is the interpretation of the bytes in the interface as shown on the left in the figure 2.8 below. The disassembled code is also displayed as a c code in the decompiled view as shown on the right in the figure below. The decompiled code is very useful because it provides a high-level view. This view is very useful for reverse engineers because this high-level view helps them to understand the code.

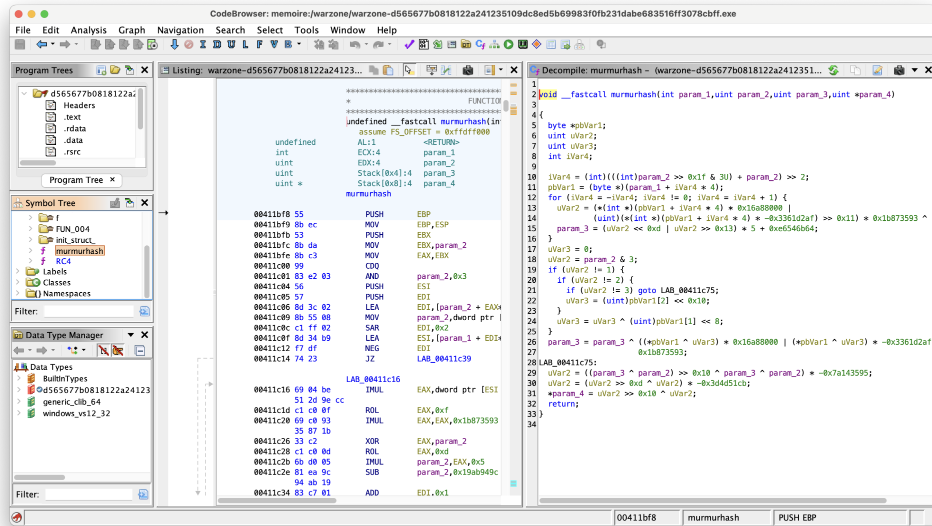

Figure 2.8: Ghidra interface [2] with the decompiled view of a function

Another very useful feature of Ghidra is the ability to create structures. As explained earlier, a lot of information is lost during the compilation process. One of the things that is lost is the structure of the code. Ghidra allows reverse engineers to recreate those structures. This is very useful because it is easier to understand the code when working with structures rather than with independent variables.

Ghidra also provides other views of the code such as the graph view that can be seen in figure 2.9. This view is useful to understand the structure of the code. It helps detect loops, if-else statements, etc. It is also an easy way to detect similarities between functions as it provides a visual representation of the code.

In this thesis, we use Ghidra to analyze binary code and confirm the results obtained with SAFE. We also use Ghidra through our experiments to analyze malware and find the address of the functions that were hooked by SAFE.

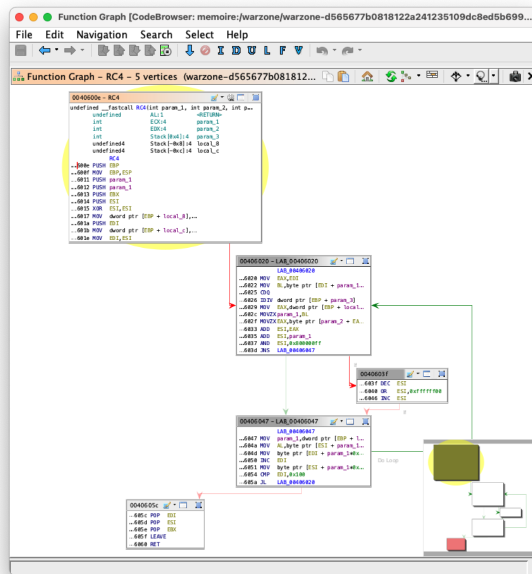

Figure 2.9: graph view of a function in Ghidra [2]

2.4.2 Radare2

Radare2[6], also known as r2, is an open-source framework designed for reverse engineering and binary analysis. It is a very powerful tool that supports a wide range of executable formats and architectures. It is built around a disassembler that generates assembly source code from machine code. Despite its powerful capabilities, Radare2 has a steep learning curve due to its command-line interface and the absence of a graphical user interface by default. However, it offers bindings for several languages such as Python, Ruby, Perl, etc. that can be used to write scripts and automate tasks. Radare2 also supports a variety of plugins that can be used to extend its functionalities.

Radare2's main features include disassembling, debugging, analyzing, patching, and scripting. In this thesis, we use Radare2 as a disassembler. The disassembled code is then used by SAFE to create embeddings of the functions. We also use Radare2 to analyze the malware and find the address of the functions that were identified by SAFE as functions that need to be hooked in the SEMA-toolchain.

Chapter 3

Contribution

In this chapter, we discuss the implementation details of the solution proposed in this thesis. We first describe how we use SAFE to create a signature for specific functions in a binary in section 3.1.1, and how we use this signature to find similar functions in other binaries 3.1.2. We then discuss how we automatically select the threshold for the cosine similarity in section 3.2. In section 4.5 we discuss how we use SAFE to find common functions in malware families. We also explain how we integrated SAFE in the SEMA-Toolchain in section 3.4 in order to replace the current hooking mechanism. The objective of this integration is to improve the hooking mechanism in the SEMA-Toolchain by replacing the pattern matching with a more efficient method both in terms of accuracy and number of signatures stored. Finally, we will discuss the challenges we faced during the implementation of our solution in section 3.5.

3.1 Finding functions in binaries

The first objective of this Master thesis is to find functions in a binary that are similar to a target function. We want to use this to improve the hooking in the SEMA-Toolchain by replacing the pattern matching. We want to match as many functions as possible that are similar to the target, but we do not want to match any functions that are not similar to the target. This is important because we do not want to hook the wrong functions and execute a SimProcedure that does not match the code. Another goal is to keep relatively small signatures as we do not want to store too much data. Finally, the function detection needs to be fast as we want to analyze a large number of binaries in a reasonable amount of time.

3.1.1 Signature creation

The first step in our approach is to create a signature for each target function. We use the radare2 framework to disassemble the binary and extract the functions. The function instructions are then passed to SAFE to create the embedding. Finally, we store the embedding in a JSON file. This first step requires to have access to the address of the function in the binary to extract the function's instructions. This step did not require a lot of changes from the original SAFE implementation but it is only suitable to compare functions in binaries that were already analyzed at least in part as we need the address of the function in the binary. In our case, it is thus only suitable to create signatures for the target functions as we do not want to analyze the binaries before running the SEMA-Toolchain.

For binaries that were not analyzed, we need to create a signature for each function in the binary and then compare the signatures against our database of target embeddings. This time instead of looking for one function in the binary, we create a signature for each function in the binary. This approach is very time-consuming because we need to create a signature for every function in the binary. For example, for the warzone sample with the sha256 hash d565677b0818122a241235109dc8ed5b69983f0fb231dabe683516ff3078cbff, it takes 58 seconds to create a signature for each function and compare them against the database of signatures. Radare 2 is able to detect 449 functions in this binary and the target database contains seven signatures.

This is too slow for this approach to be used in the SEMA-Toolchain. We need to find a way to speed up the process. The process can be divided into two steps. The first step is to create the signature for each function in the binary and store it in a temporary file. This step is very slow and can be done before running the SEMA-Toolchain as a preprocessing step. The second step is to compare the signature stored in the temporary file against the signatures of the database of target functions. This step is much faster and can be done during the execution of the SEMA-Toolchain. Furthermore, if enough memory is available, the signature can be stored permanently in a database instead of a temporary file. This way the signatures can be reused when the binary is analyzed multiple times saving time in the long run.

3.1.2 Signature matching

The second step in our approach is to compare the signature of the target function with the signatures of the functions in the binary. We use the cosine similarity

to compare the signatures. The cosine similarity is defined as the cosine of the angle between two non-zero vectors. The output of the cosine similarity is bounded between zero and one where zero means that the two vectors are orthogonal to each other, meaning that they do not have anything in common, and one means that the two vectors are identical. We do not want to know if the two functions are identical but if they are similar. We thus need to find a threshold that will allow us to determine whether two functions are similar.

The advantage of the cosine similarity is that it is fast to compute. This is important because this part cannot be computed before running the SEMA-Toolchain. For example, it only takes four seconds to compare the signature of the target function with the signatures of the functions in the binary. This is much faster than computing the signature for each function in the binary. The binary and the database of target functions used are the same as in the previous example.

3.2 Automatic threshold selection

As we have seen in the previous section, we need to find a threshold that allows us to determine whether two functions are similar. We want to find a threshold that will allow us to match as many of the functions that are similar to the target as possible but we do not want to match any functions that are not similar to the target.

The first attempt was to find a fixed threshold that would work for all the functions. We quickly realized that it was not possible to use the same threshold for all the functions. Indeed, the ideal threshold is very different for each function. This means that if we choose a threshold that is ideal for a given function, this threshold might be too high for another function and we might miss some matches. On the other hand, the threshold might be too low for another function and we might match functions that are not similar to the target. This approach is thus not suitable for our needs and was quickly discarded.

The second attempt consisted of trying to find a fixed threshold for each binary family. This approach seemed more promising because it is more flexible than the previous one and because binaries of the same family are more likely to have similar functions. However, we were confronted with the same problem as before as binaries of the same family contain a wide variety of functions and the ideal threshold is different for each function. A closer look at the different signatures for the same function in different binaries of the same family showed that different signatures for the same function also have different ideal thresholds. This approach

was discarded as well.

The final solution that we implemented is to use a threshold for each signature. We first began by selecting a threshold manually by looking at the cosine similarity between the signature of the target function and the signature of the other functions of binaries from the same family but this is a slow process. We then decided to select automatically the threshold. To select automatically the threshold, we use a dataset of binaries of the same family as the binary to which belongs the target function. We calculate the cosine similarity between the embedding of the target function and the embeddings of the functions of the dataset. For each binary, we select the two functions with the highest cosine similarity and temporarily store them by threshold. We then select the threshold as the value above the second threshold with a nonnull number of functions. We assume that the first threshold with a nonnull number of functions only contains functions that are similar to the target function and that the second threshold with a nonnull number of functions contains at least some false positives. This approach is much faster than the previous one and allows us to select the threshold automatically.

3.3 Finding common functions in malware families

The second objective of this Master thesis is to identify common functions in malware families. This is an important task because it can help us save time when trying to identify to which family a malware belongs. This task is very similar to the first one. The only difference is that we do not compare a binary against a target function but we compare a binary against several other binaries. We begin by creating signatures for each of the functions of the first binary and store those signatures. We create a signature then for each of the functions of the other binaries and compare them against all of the stored signatures. If a signature is similar to one of the stored signatures a counter is incremented. If a signature does not match any of the stored signatures, we add it to the database of signatures and set the counter to one as the function this signature belongs to might be similar to other functions in binaries that were not yet analyzed.

In this case, we do not know the threshold that we should use for any of the signatures. We cannot compute a custom threshold for each signature as we did in the previous section because instead of having a few target functions for which we needed a threshold, we now need a threshold for each function in the binary. We

thus choose to use a fixed threshold for all of the signatures. We do not want any false positives but we can miss some matches as the main objective is not to use this threshold as the final threshold when looking for functions but to facilitate the process of finding common functions in malware families. We thus choose a threshold of one.

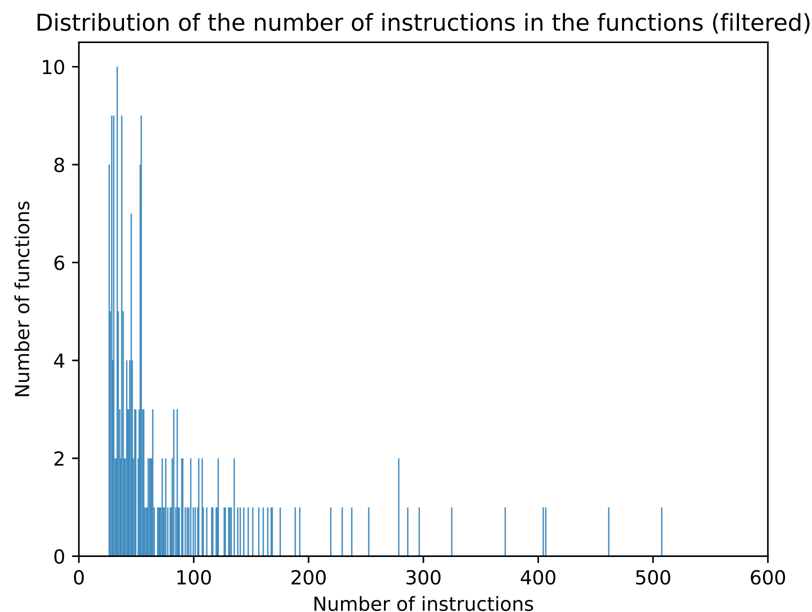

Figure 3.1: Distribution of the number of instructions in the functions of the warzone sample with the sha256 hash d565677b0818122a241235109dc8ed5b69983f0fb231dabe683516ff3078cbff

After the first few attempts, we realized that the process was too slow due to the number of functions that were analyzed and the number of signatures that were stored. We thus decided to plot the distribution of the number of instructions in the functions as shown in Figure 3.1. We realized that most of the functions were very small and would probably not contain any interesting code. To speed up the process, we decided to filter out the functions that had less than 25 instructions. This reduced the number of functions to be analyzed by 47% bringing the number of functions to be analyzed down from 449 to 236 in the warzone sample that was used in the previous section.

The output of this process is a list of signatures along with the number of times they were matched. We want to keep only the signatures of the functions

that are interesting to analyze. We choose two criteria to determine whether a function is interesting or not to analyze. The first criterion is the number of times the signature was matched. We want to keep functions that are common in the malware family. The second criterion is the number of instructions in the function. We prefer to analyze larger functions as they are more likely to contain interesting code and be complex enough to slow down the symbolic execution. To respect those two criteria, we sort the list with the formula $score = n \times m$ where n is the number of instructions and m is the number of matches. We then keep the top 10 functions. Those functions are then analyzed to determine whether they are interesting or not. We also try to find a threshold that allows us to match similar functions in other binaries of the same family.

3.4 Integration in SEMA-Toolchain

The final objective of this Master's thesis is to integrate SAFE in the SEMA-Toolchain. The SEMA-Toolchain already contains several additional functionalities such as the current hooking mechanism that are organized in different plugins. The simplest way to integrate SAFE in the SEMA-Toolchain was to create a new plugin that would use SAFE to find the functions to hook. This plugin would then replace the current hooking mechanism.

To make the integration as simple as possible, we decided to create a new plugin that would require almost the same inputs as the current hooking mechanism. This way the main code of the SEMA-Toolchain would need to be modified as little as possible. The current hooking mechanism consists of looking for a predefined list of byte patterns in the text section of the binaries. If a match is found, the function is hooked. The new plugin would use SAFE to find the functions to hook. The current hooking mechanism first requires an initialization step just before the main hooking mechanism where a file descriptor for the binary is given and the architecture of the binary is determined. Then the functions are hooked in a second function which requires a state, the project manager and the module containing the SimProcedures. The new plugin brings very small changes to the main code of the SEMA-Toolchain. First, the initialization step is now done before the beginning of the symbolic execution and consists of creating and storing a signature for each function in the binary to be analyzed. The second change is the arguments needed as input by the main hooking function which are now the path to the binary, the project manager and the module containing the SimProcedures. The plugin then hooks the functions in the project manager.

The integration of SAFE into the SEMA-Toolchain required some changes to our tool. The hooking mechanism in the SEMA-Toolchain required the address of the function that needed to be hooked and the length of the function. This information is not available in the signature of the function as it can change from one binary to the other. We thus modified the script that decompiles the binary to return the address of the function, the length of the function and the last instruction of the function. This is important because if the last instruction is a return, it must be removed from the function's length.

3.5 Challenges

During the implementation of this tool, we faced several challenges. The first one was to create a suitable way to create a signature for each function. My first attempt was to use the decompile view of Ghidra to recreate the function and compile this function alone in a new binary. We then tried to extract a signature from this new binary. This approach gave very poor results. The best result we got was a cosine similarity of 14% which is very low. The alternative that we chose was to find the address of the function in the original binary and extract the signature directly from the binary. This approach gave much better results with a cosine similarity that reached up to 100%.

The second challenge that we encountered was a memory issue. We noticed that after some time the process terminated with an out-of-memory error. After some investigation, we realized that a lot of radare2 processes kept running even after the process was finished. We thus added some code to kill the radare2 processes after the process was finished. This solved the issue.

We came across one last challenge when analyzing a wabot sample (sha256: 9ca786e161159f6fb7aead62db463e5f2b42c7a3c3f6e1126d9462ca039cca47). We noticed that some functions were not detected as functions by radare2. In this case, we wanted to create a signature for the function at the address 0x402890 which writes the weed leaf in a file but no function is found at this address. Other decompiler tools such as Ghidra or IDA Pro do not detect the function either. We could not find a solution to this problem as specifying the address of the function and the length of the function to create a target signature is not of any use as the function will probably not be detected in new binaries.

Chapter 4

Experimental Results And Analysis

In this chapter, we present the results of our experiments. We evaluate the performance of our function identification tool and our threshold selection algorithm on a set of real-world malware, including the Warzone malware and the Satan malware. We selected these malware samples because they have already been analyzed in the SEMA-Toolchain where several functions were slowing down the analysis. In order to improve the efficiency of the analysis, SimProcedures were added to the SEMA-Toolchain. In section 4.1, we evaluate the performance of our function identification tool on real-world malware. We then proceed to identify the optimal threshold for those functions in section 4.2. In section 4.3, We demonstrate that our tool can be used to find common functions in malware families. Those functions can then be used to identify the malware family of a binary. We also present our first use case, which is improving the performance of the SEMA-toolchain by identifying functions in the SEMA-toolchain in section 4.4. In this section, we evaluate the impact of our function identification tool on the performance of the SEMA-Toolchian. Finally, we present our second use case, which is the identification of functions in binaries compiled with different compilers in section 4.5. For this last experiment, we use the GonnaCry malware.

4.1 Function Identification In Binaries

The first experiment is designed to assess the performance of our function identification tool in a real-world malware context. Specifically, The objective is to identify functions within Warzone malware and Satan malware. The Warzone malware samples were divided into two groups: the samples compiled with Visual Studio

and the samples compiled without Visual Studio. This distinction is made because when a binary is compiled with Visual Studio, the compiler adds some additional functions that are not present in a binary compiled with another compiler. Furthermore, some instructions are also added which makes the identification of functions more challenging. It is therefore of great benefit to be able to identify those binaries.

In the next sections, we compare the performance of our tool with that of pattern matching in terms of the number of signatures required to identify a function in all samples.

4.1.1 Warzone

The first set of samples to be analyzed is the samples of the Warzone malware compiled without Visual Studio. A total of 66 samples will be analyzed, with the objective of identifying four distinct functions within each sample. The samples were obtained from the SEMA-Toolchain. They can be found in the folders database/malware/warzone and database/malware/warzone3. Additionally, they can be downloaded from malwareBazaar [5]. The objective is to identify the three functions that were hooked by the SEMA-Toolchain. These include the murmurhash function, which is a non-cryptographic hash function; the find_start function, which is used to find the offset of the start of the PE file in memory; and the copy function, which is used to copy data. Additionally, we want to identify the RC4 function, which is used for encryption purposes.

function	number of signatures with pattern matching	number of signatures with SAFE
RC4	3	3
murmurhash	2	2
find_start	6	3
copy	2	2

Table 4.1: Number of signatures needed to identify functions in Warzone malware with pattern matching and SAFE

Table 4.1 provides a summary of the results of this initial experiment. The SAFE algorithm identifies functions with an equal number or lower number of signatures compared to the pattern-matching algorithm. It can be observed that for three out of the four functions, the number of signatures required to detect the functions with SAFE is identical to that required with pattern matching. This suggests that the

SAFE algorithm is unable to identify functions with a smaller number of signatures than the pattern-matching algorithm. Nevertheless, it can be observed that for the find_start function, SAFE requires three signatures, whereas the pattern-matching algorithm necessitates six signatures. This is a significant improvement. Furthermore, in the SEMA-Toolchain, only four out of the six signatures were used to identify the find_start function. Moreover, thanks to SAFE, we were able to find out that three of the binary samples contained the find_start function and the SEMA-Toolchain did not detect it. Two new patterns were needed to cover those three samples with pattern matching while no new signature was needed with SAFE.

Further analysis of the three functions requiring the same number of signatures by SAFE and pattern matching reveals significant differences. For example, Figure 4.1 illustrates the graph views of the RC4 function in the Warzone malware. We can see that the three graphs are very different, despite the functions they represent exhibiting the same functionality. Upon analysis of the code, it can be assumed that the three functions were not compiled from the same source code. As a consequence of SAFE's definition of similarity, two functions are considered similar if they are compiled from the same source code. This limitation explains why, in this case, the number of signatures needed to identify the functions is the same with SAFE and with pattern matching.

We also observe that functions that require several patterns but only one signature with SAFE are nearly identical. For example, the find_start function requires six signatures with the pattern-matching algorithm. Upon examination of the patterns, we notice that three of the signatures are identical with the exception of the addresses of the functions that are called. Additionally, two other signatures are also identical with the exception of the addresses of the functions. Since SAFE standardizes the instructions by replacing all base memory addresses with the symbol "MEM" and all immediate values above 5000 with the symbol "IMM", the patterns that only differ by the addresses of the functions are identical to the SAFE model after standardization. This is why we only need three signatures to identify the find_start function with SAFE.

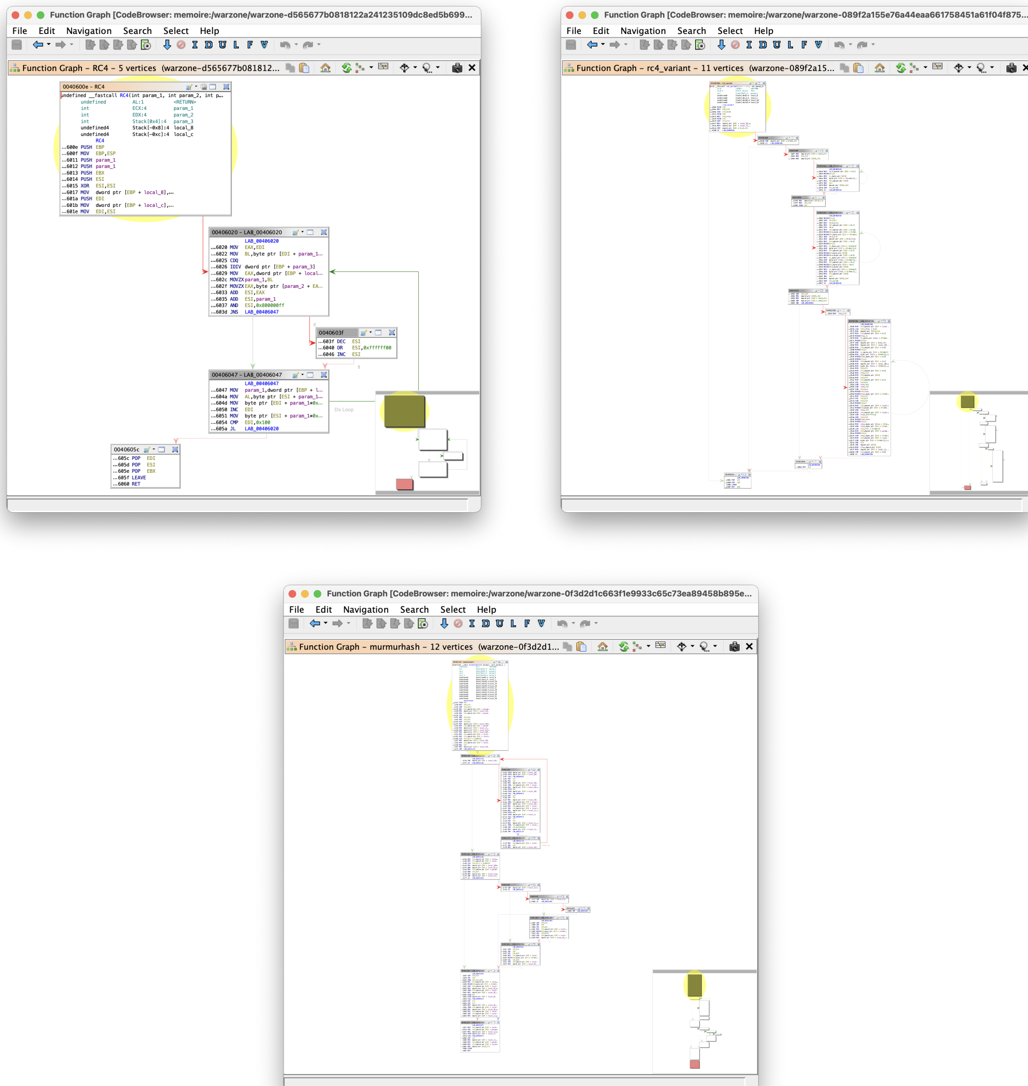

Figure 4.1: graph views of the RC4 function in the Warzone malware

4.1.2 Warzone Compiled With Visual Studio

We now analyze the samples of the Warzone malware compiled with Visual Studio. A total of nine samples were analyzed. The samples can be found in the SEMA-Toolchain in the folder src/databases/malware/warzone2, or alternatively, they can be downloaded from malwareBazaar [5]. The objective is to identify the binaries compiled with Visual Studio by identifying the functions added during the compilation process.

To identify the binaries compiled with Visual Studio, we are going to use the ___security_init_cookie function. This function is added by the Visual Studio compiler before the main function. A total of five distinct patterns were identified in the nine samples. With SAFE, it is possible to identify the ___security_init_cookie function in all 9 samples using only two signatures. Upon further analysis of the results, we notice that three of the patterns are almost identical, with the exception of the addresses of the functions that are called. The two last patterns are also almost identical with the exception of the addresses of the functions that are called. This is why we can identify the ___security_init_cookie function in all nine samples with SAFE using only two signatures.

4.1.3 Satan

We now analyze the samples of the Satan malware. A total of nine samples were analyzed. All samples were downloaded from malwareBazaar [5]. The objective is to identify the CRC32 function, which is used for encryption purposes. Pattern matching requires four signatures to identify the CRC32 function across all samples, whereas SAFE requires only one signature, representing a significant improvement.

This observation is consistent with the results of the previous two experiments. The functions that require multiple signatures for pattern-matching but only one signature with SAFE are nearly identical. In this instance, all four patterns are identical, with the exception of the addresses of the functions that are called.

4.1.4 Conclusion

This experiment has demonstrated that SAFE is capable of identifying functions in malware samples with the same number or fewer signatures than pattern matching. It has been shown that sometimes the number of required signatures decreases significantly, usually when the underlying implementations are similar. Furthermore, the results demonstrated that SAFE can be employed to identify binaries compiled with Visual Studio, which is a noteworthy finding given that these binaries are typically more challenging to analyze.

4.2 Threshold Selection

The second experiment is designed to assess the performance of our threshold selection algorithm on a set of real-world malware. In this experiment, the objective is to identify the optimal threshold for several functions of the Warzone malware and The Satan malware. The samples used in this experiment are the same as the ones used in the previous experiment. The functions to be analyzed are identical to those used in the previous experiment.

The results of this experiment are presented in Table 4.2. We can see that the threshold selection algorithm is able to find the optimal threshold for each function. We can see that the threshold varies between functions and even between signatures of the same function. This corroborates our initial expectations and observations when the threshold was selected manually.

function's signature	computed threshold
murmurhash1	0.98
murmurhash2	0.98
find_start1	0.95
find_start2	0.95
find_start3	0.95
copy1	0.93
copy2	0.98
CRC32	0.93
RC4_1	0.95
RC4_2	0.95
RC4_3	0.98

Table 4.2: threshold associated with each function's signature

Upon analysis of the results of the threshold selection algorithm, we notice that the selected threshold represents the optimal threshold, as it is the lowest threshold that does not generate any false positives. For example, we can see that the threshold selected for the copy1 signature is 93%. A lower threshold would result in the generation of false positives. Figure 4.2 illustrates that other functions are matched with the copy1 signature when the threshold is set to 90%.

```
"1": [
    "acd0a278ad8f069876948274d6d25f07d6a4235816f9305bf54b2e2af3a401df.exe:0x405f10",
    "83fbc8a0d3401b8608393235eb841e6460617f335df60175844419090340d322.exe:0x406056",
    "3bafaa53c171d4db2f5b4f70854074ac6f94603fa05adb345c8eda3ecb57a64d.exe:0x406056",
    "a630663a4107d39f058a44be1e18983ba525a102ac3de6385d334e42286a3b16.exe:0x406056",
    "d456904e3307857acb2bdfd775ae01a6eae2aeb97e272cf7c03ff908d11078f1.exe:0x405f10",
    "48ec64cdc006748cb30b5f89d324cc32de2f801306e756c8123845eb7f7c6314.exe:0x405f10",
    "afdbfdbff1e5e9fc699238760938bf623bafbf956fc4e9b2821f49bdc64d484f.exe:0x405f10",
    "a0b47f3da2bfe1338fdd4eb4ee469ceeff27fa468d01ced504b65a47a771ed8a.exe:0x406056",
    "0fb4fd33e3c5368794c0f8e8d5556ef25f1fc808e8db64c9b45e7949e6ee4736.exe:0x406056",
    "68191d4ba2ad20bf53986dfc5db193397a6bead2c9342078d9a34475230659ef.exe:0x406056",
    "4f37c4e79d72c48098834406da459352c8ed7c3f992aa3eb4bb1f2f7f57142a5.exe:0x406056",
    "f602d87cc61626a3919c3a7d84f1d2e6f90c95d526cf9db636c8c60edd7f1523.exe:0x406056",
    "0bb084679cd7cc438060f3767431e46a6ca4b45cead37ca807fb60856ef811bc.exe:0x406056",
    "a71b96d3414fc0bd86d7701a20fd3f853e70597888a3f0fc0944ba2bd8efea09.exe:0x405f10",
    "0b798bdaee2272a2229129516db57688a7152cc22f2cf7f5fc8abafa91d5d44c.exe:0x405f10",
    "64421ab9070f46618ed715eabad6773d34284c31fb9da36fa4420cab700edfb2.exe:0x405f10",
    "023b25919ded48abf8ce886c88c1f4d95bd49bc201e94117007e6e1080c2696e.exe:0x405f10",
    "82428eab38bb8231eed298544e18c6cd2540ba0eb3d119ea6acec839f2ed3505.exe:0x406056",
    "e6c579b3e2afee4873d9ccc979939eb3f3fe0401a56ada7424eb0da2231a45a0.exe:0x406056",
    "a6c102c46b5af5af454e7167fd7d423bae97b3c65519ea8e1ca052ea9736e9dd.exe:0x405f10",
    "d565677b0818122a241235109dc8ed5b69983f0fb231dabe683516ff3078cbff.exe:0x405f10",
    "86eaa1914e20f44397eefb399da3f9b35b10c06c1fadf5f8d8877f4e24c27227.exe:0x406056",
    "9a3893833dc83a8411169661d6a3c7789c0d23f8240dc27ccce8ec1d148bf529.exe:0x406056",
    "0fdb0b3d02150d8377b722507554cab890ee8e8fc3a528a5d31a5ff57e8a70e7.exe:0x405f10",
    "d1b0015f92c9dae17c744239340fe4035580612c80fd24cf7a8f04cf5a6b74a3.exe:0x4058d4",
    "0c177d5b5acb632686c85983d3db56ab270d3bf80011cb2a82294fffacb049ca.exe:0x406056",
    "288aaf8f0ada25a39dc9cbdec687dbd0c79cdc9d61f748427523ed86a91bc143.exe:0x406056",
    "c2737f26a23c6d9dcdd21f052f85414c3a2b92455df9173c7a9874a52f438233.exe:0x405f10",
    "e8458f7ed5b1a78f751831bb1837da7a84f33c8a67579d5928d8e3f01534a582.exe:0x406056",
    "35da24f2eaaf244a17ad7e8693f679ec31ce04b09b001ca0389c2da94dcea73c.exe:0x405f10",
    "bde560e8a84bef5b1422ef285963f29f653c683f96e3b83c40e81c41fa266d2f.exe:0x405f10"
],
"0.98": [],
"0.95": [],
"0.93": [],
"0.9": [
    "2fb55700d343afcad180486bedddb4ce8a632d11cbbde696d8db7a165543ae90.exe:0x40126d",
    "acd0a278ad8f069876948274d6d25f07d6a4235816f9305bf54b2e2af3a401df.exe:0x40102c",
    "83fbc8a0d3401b8608393235eb841e6460617f335df60175844419090340d322.exe:0x40102c",
    "948484608e00c0e3d315b277f55a9ebbb73210297e532daeac3cec271b4ffe85.exe:0x40126d",
```


Figure 4.2: functions matched with the copy1 signature with different thresholds

4.3 Finding Common Functions In Malware Families

In the third experiment, we attempt to identify common functions in malware families. This approach can be used to classify binaries and to identify the malware family to which a given binary belongs. The Warzone malware samples that were used in the previous experiments will be used in this experiment.

The algorithm identifies the ten functions that are both the most common and have the highest number of instructions. Those functions and their respective number of instructions are presented in table 4.3. It can be observed that only one function is found in the majority of the samples when the threshold is set to 100%. For each of those functions, an attempt is made to identify the optimal threshold. Lowering the threshold may assist in the identification of more similar functions.

The first thing that we notice when analyzing the results is that lowering the threshold effectively increases the number of similar functions found. Furthermore, we notice that the first two functions are similar. They have a cosine similarity of 0.93. Upon lowering the threshold to 93%, we can see that only functions that are similar to the target function match the embedding. The target function is

md5 of the binary and address of the function	number of match with a threshold of 100%	number of instructions
d8970bb47eac252ab91e347e6a77a9d2 0x40847a	33	1522
226842d0853fb16bf3e5379180935ca2 0x406297	31	1523
d8970bb47eac252ab91e347e6a77a9d2 0x40e331	32	604
d8970bb47eac252ab91e347e6a77a9d2 0x40f480	32	571
d8970bb47eac252ab91e347e6a77a9d2 0x40d379	32	531
d8970bb47eac252ab91e347e6a77a9d2 0x405f6c	33	502
d8970bb47eac252ab91e347e6a77a9d2 0x40cd01	32	491
d8970bb47eac252ab91e347e6a77a9d2 0x40fb4b	32	479
226842d0853fb16bf3e5379180935ca2 0x40ba00	30	507
226842d0853fb16bf3e5379180935ca2 0x40b67e	63	219

Table 4.3: number of common functions matched in the Warzone malware with a threshold of 100% and number of instructions in those functions

detected in 64 out of the 66 samples. This means that the function can be identified in 97% of the samples without any false positives. It is thus an excellent candidate to determine the malware family of a binary. Furthermore, this function has more than 1500 instructions. It is thus a pretty large function which also helps to prevent false positives as smaller functions can be more easily confused with one another. The results of this experiment are presented in Table 4.4. The results demonstrate that the majority of the identified functions exhibit a high degree of similarity across the vast majority of the samples analyzed.

We also notice that the functions matched by the same embedding can sometimes have a different call graph. For example, the third function has different call graphs in different samples. Figure 4.3 shows the graph views of the function identified in the warzone samples with the md5 hash values d8970bb47eac252ab91e347e6a77a9d2 and 226842d0853fb16bf3e5379180935ca2. The functions can be found at the address 0x40f480 in the first sample and at the address 0x40c4a8 in the second sample. We can see that the call graphs are very different. This is noteworthy, as it demonstrates that when the functions are complex enough, SAFE is capable of identifying

md5 of the binary and address of the function	threshold	number of functions matched with the new threshold
d8970bb47eac252ab91e347e6a77a9d2 0x40847a	93%	64
226842d0853fb16bf3e5379180935ca2 0x406297	93%	64
d8970bb47eac252ab91e347e6a77a9d2 0x40e331	95%	32
d8970bb47eac252ab91e347e6a77a9d2 0x40f480	95%	62
d8970bb47eac252ab91e347e6a77a9d2 0x40d379	95%	64
d8970bb47eac252ab91e347e6a77a9d2 0x405f6c	93%	63
d8970bb47eac252ab91e347e6a77a9d2 0x40cd01	98%	66
d8970bb47eac252ab91e347e6a77a9d2 0x40fb4b	95%	66
226842d0853fb16bf3e5379180935ca2 0x40ba00	98%	32
226842d0853fb16bf3e5379180935ca2 0x40b67e	95%	66

Table 4.4: optimal threshold and number of common functions in the Warzone malware matched with the new threshold

them even if they are different. This represents an improvement on the previous experiments as previously only identical functions could be identified, with the exception of the addresses of the functions that were called.

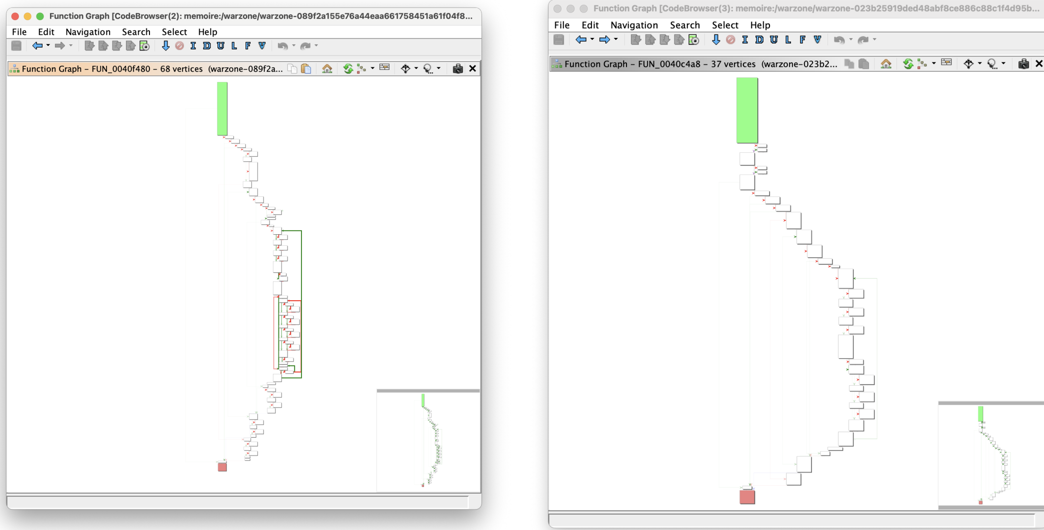

Figure 4.3: graph views of the functions found in the Warzone malware

4.4 Use Case1: SEMA-Toolchain

Now that we have evaluated the performance of our function identification tool and our threshold selection algorithm, we can use them to improve the performance of the SEMA-Toolchain. In this experiment, we are going to try to identify the functions that are hooked in the SEMA-Toolchain to speed up the analysis and evaluate the performance of the SEMA-Toolchain with and without using SAFE. We are going to use the Warzone malware samples that were used in the previous experiments.

We want to evaluate the performance of the SEMA-Toolchain with and without using SAFE. The metrics that we use are the number of signatures used to match the functions. We are also going to compare the number of syscalls found in the Warzone samples by the SEMA-Toolchain during the symbolic execution. This is often a good metric to evaluate the performance of the SEMA-Toolchain as the number of syscalls found is directly related to the depth of the analysis of the binary.

For this experiment, we ran two symbolic executions on all 66 warzone samples with the SEMA-Toolchain. The first one uses pattern-matching to identify functions using the patterns that were already present in the SEMA-Toolchain. The second one uses SAFE to identify the functions. We use the same search strategy for both executions. In this case, we used the CDFS strategy. All of the other options were set to their default values. These options contain a timeout which ensures that

each analysis is stopped after the same amount of time, in this case 1000 seconds.

The first metric that we use to evaluate the performance of our implementation is the number of signatures needed to hook the functions. For the symbolic execution using pattern-matching, the SEMA-Toolchain uses 9 different signatures to identify the three functions that are hooked in all of the binaries. We discovered that 11 signatures would be needed to detect the three functions in all the binaries but two of the signatures for the find_start function were missing. The function was thus not detected in three of the binaries resulting in a loss of performance. To identify the same functions with SAFE in the same samples, we only need 7 signatures. This is already a significant improvement.

The second metric that we use to evaluate the performance of our implementation is the number of syscalls found in the Warzone samples by the SEMA-Toolchain. The difference in the number of syscalls found by the SEMA-Toolchain with and without using SAFE is presented in Figure 4.4. The y-axis represents the number of syscalls found and the x-axis the samples. The 6 first characters of the sha-256 of the samples are used as labels. We can see that the number of syscalls found is very similar for most of the samples. This shows that considering this metric, the performance of the SEMA-Toolchain with and without using SAFE is very similar. However, we can see that two samples, the one with 2c9adbeac37afa788d0fbdd01fba9d91 as md5 and the one with ac8a21c224860f80c8ef0b3edda4cd6a as md5, have a large difference in the number of syscalls found with and without using SAFE. When further analyzing the results, we found out that SAFE was able to identify a find_start function in the two samples where it was not identified by the pattern-matching algorithm because the pattern was missing. The third sample in which the find_start function was not identified by the pattern-matching algorithm cannot be found in the results as the analysis with the SEMA-Toolchain is interrupted by an error before the end of the analysis both when using pattern matching and SAFE to hook the functions. This shows that SAFE is more robust than pattern-matching. It is very important to have a robust function identification tool because the performance of the SEMA-Toolchain is directly linked to the number of functions that are hooked. In this case, for both binaries, the SEMA-Toolchain found 367 syscalls in total when using patten-matching while it found 2250 syscalls when using SAFE.

In conclusion, we have shown that our implementation is able to improve the identification of functions in the SEMA-Toolchain. First by reducing the number of signatures needed to identify the functions and second by identifying functions that were not identified by the pattern-matching algorithm. We have also shown that the performance of the SEMA-Toolchain is very similar with and without

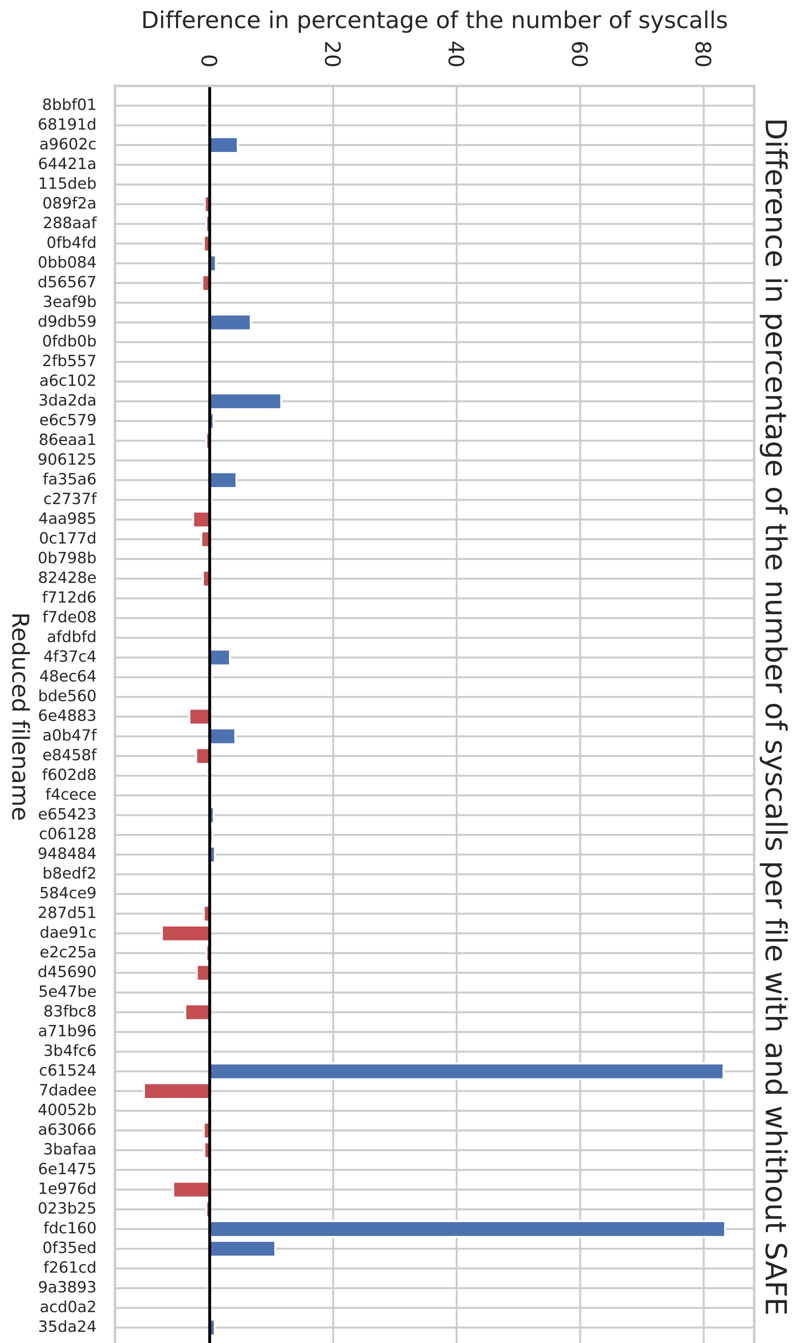

Figure 4.4: Number of syscalls found in the Warzone samples by the SEMA-Toolchain with and without using SAFE

using SAFE considering the number of syscalls found. Furthermore, we have shown that SAFE can significantly improve the performance of the SEMA-Toolchain by identifying functions that were not identified by the pattern-matching algorithm.

4.5 Use Case2: Finding Functions In Binaries Compiled With Different Compilers

This section will present our final experiment. Our previous experiments demonstrated that SAFE was only able to identify functions if they were identical except for the addresses of the functions that were called. This phenomenon can be attributed to the fact that the SAFE model was trained to identify functions in binaries compiled from the same source code. The objective of this section is to verify this hypothesis by attempting to identify functions in binaries compiled from the same source code but with different compilers and compiler optimization levels. For this experiment, we will use the GonnaCry malware and compile it ourselves with different compilers and optimization levels. The source code of the GonnaCry malware can be found on GitHub [3]. The source code was compiled using the GCC and Clang compilers with the -O1, -O2, and -O3 optimization levels. To enhance the experimental rigor, the binaries were also stripped. This is a common technique used by malware authors to make the analysis of the binary more challenging.

The first step of this experiment is to determine which functions we are going to try to identify in the binaries. The SEMA toolchain does not include any functions that are linked to the GonnaCry malware. It is therefore necessary to determine which functions are worth identifying. We will use the tool from the previous experiment to find pertinent functions that are common in the generated binaries.

Upon examination of the results of this first part of the experiment, we immediately notice that none of the functions compiled with clang have a cosine similarity of 1 with any of the functions compiled with gcc. This is noteworthy because it demonstrates that, despite source code being identical, the compilers generate distinct binary codes. We then see that the functions are fairly large and have between 68 and 151 instructions. We are now interested in finding the optimal threshold for those functions to try to identify more similar functions.

The results of the threshold selection are presented in Table 4.5. In this experiment, the threshold selection algorithm was not employed, as only six binaries were available for analysis. This means that we do not have a big database to

function	threshold	number of functions matched	number of instructions
gonnacry_clang_O1:0x1b40	88%	6	138
gonnacry_gcc_O2:0x1f50	88%	6	151
gonnacry_gcc_O2:0x18c0	83%	6	115
gonnacry_clang_O1:0x1370	89%	6	68
gonnacry_clang_O2:0x15b0	75%	6	92
gonnacry_clang_O2:0x1e40	77%	4	91
gonnacry_gcc_O2:0x22f0	91%	3	90
gonnacry_clang_O2:0x1e10	100%	2	83
gonnacry_gcc_O2:0x2450	100%	2	81
gonnacry_gcc_O2:0x1540	89%	6	79

Table 4.5: results of the threshold selection algorithm for 10 most common functions in the GonnaCry malware

compare the functions with. Consequently, the threshold was selected manually as the first threshold at which all similar functions were identified. We can see that six out of the ten functions are detected in all of the binaries. This demonstrates that SAFE is capable of identifying similar functions, even when they have been compiled with different compilers and optimization levels. The used thresholds are a lot lower than the ones used in the previous experiments. This is a sign that the SAFE model differentiates the functions more easily when the functions are compiled from the same source code.

To verify whether the chosen threshold is high enough and that no false positives will be found, we used other binaries from different families and verified whether other functions were matched. We used binaries from the Warzone family. We found out that no other function was matched. The maximal cosine similarity observed was 0.65 which is below all of the thresholds that we used. This indicates that the chosen thresholds are sufficiently high and that no false positives will be identified.

Furthermore, it has been observed that the efficacy of SAFE is enhanced when the functions are compiled from a single source code. It is capable of identifying functions that are similar but not identical, despite the fact that the functions are not particularly complex and do not have a lot of instructions. Upon examination of the control flow graph of similar functions, we can see that they are very different. This can be seen in Figure 4.5. The function on the left comes from a binary compiled with gcc with optimization level 2 and the function on the right comes

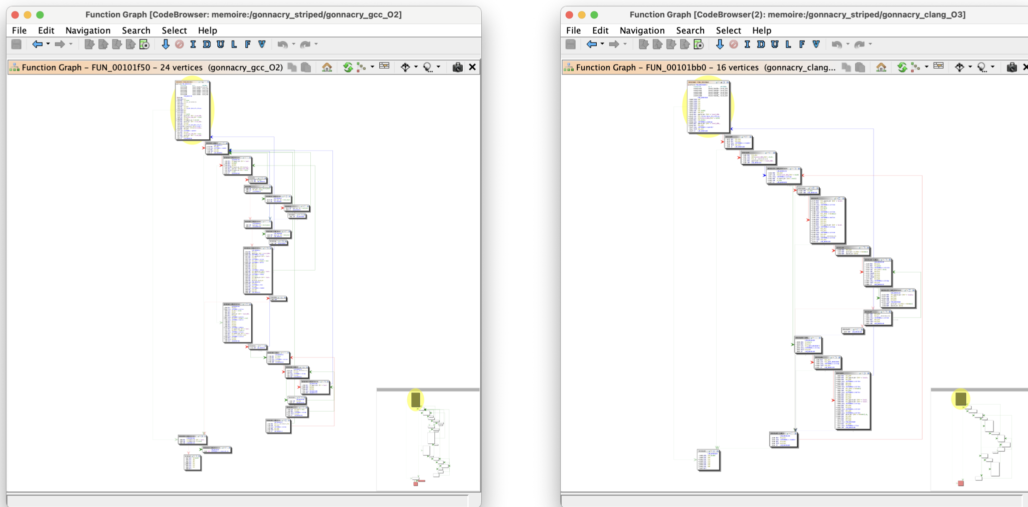

Figure 4.5: control flow graph of similar functions in the GonnaCry malware on the left compiled with gcc with optimization level 2 and on the right compiled with clang with optimization level 3

from a binary compiled with clang with optimization level 3. We can see that the two graphs are very different. This is interesting because it shows that SAFE is able to identify functions that are similar but not identical even when the functions are not particularly complex.

Chapter 5

Limitations And Future Work

The first limitation of this work is that functions are only identified as similar if they are compiled from the same source code. This is due to the manner in which the model was trained. The dataset used to train the model was constructed in such a way that two functions are considered similar only if they were compiled from the same source code. In this work, we would like two functions to be considered similar if they are semantically similar, which implies that they have the same functionality. In order to achieve this, it would be necessary to retrain the model with a dataset of functions that have the same functionality but different implementations. The main challenge of this work is to find or generate a big enough dataset to train the model. Two potential solutions were considered for the creation of this dataset. One potential solution is to use existing datasets such as the Lumina dataset from IDA[4] or the OpenFunctionID database from Ghidra [16]. The second approach would be to create a dataset by using the work of Bastien Wiaux and Arnold Gauthier [31]. They developed a mutation tool for binaries that is specifically designed to obfuscate malware. This tool could be used to obfuscate functions in malware that we compiled ourselves. This approach would enable the creation of a dataset comprising functions exhibiting the same functionality but differing in their implementation.

The second limitation of this work is that the model is only able to detect functions that are recognized as functions by radare2. To illustrate, the function responsible for writing the weed leaf in the Wabot malware is not identified as a function by radare2, rendering it undetectable. This is not a limitation of the model itself but rather of the tool used to extract the functions. This limitation is new as the previous function detection tool solely relied on the identification of a specific byte pattern within the binary to detect functions and did not rely on a decompiler. This limitation is challenging to overcome as other tools, such as Ghidra and IDApro, also fail to recognize this function as a function. One potential solution would be to use the Angr framework to extract the functions

from the binaries. Angr is a symbolic execution engine that can be used to extract functions from binaries. Furthermore, the Angr framework is already used in the SEMA-Toolchain for its symbolic execution engine. Using the Angr framework to extract the functions would, therefore, eliminate the dependency on radare2 and simplify the toolchain.

Another possible improvement is the manner in which the embeddings of the processed binaries are stored. Currently, the embeddings are stored by filename and address. Consequently, if the same function is present in multiple binaries, the signature of the function will be stored multiple times. This approach is not optimal and could be improved by using a hash of the bytes of the function as a key. This approach ensures that the signature of a function will only be stored once and can be reused if the function is present in multiple binaries. Furthermore, the process of computing a hash is relatively straightforward and will not significantly impede the overall speed of the system.

Chapter 6

Conclusion

In this thesis, we have highlighted the binary similarity problem and the challenges it presents. We have then conducted a review of several solutions proposed in the literature that address this problem. The thesis then presented SAFE, a novel approach to detecting similar functions in malware and its different components. It also explained how SAFE was integrated as a plugin in the SEMA-Toolchain.

This thesis then demonstrates that SAFE is able to accurately detect similar functions in different malware families. Furthermore, we showed that SAFE requires at most the same number of signatures as the number of patterns needed for pattern-matching. Subsequently, the results of our threshold selection algorithm were presented, demonstrating that the selected threshold is accurate. We then showed that our integration of SAFE in the SEMA-Toolchain can enhance the detection of functions and the hooking process. Furthermore, it has also been shown that, due to the enhanced robustness of SAFE compared to pattern matching, our plugin can sometimes improve the performances of the symbolic execution by hooking functions that would not have been found through pattern matching.

Additionally, We showed that SAFE can assist in the identification of interesting functions commonly found among malware families. Finally, we demonstrated that the performance of SAFE is enhanced when processing binaries compiled from the same source code but with different compilers and compiler optimizations.

Finally, we discussed the limitations of our work and shared our thoughts and suggestions on the next challenges and how to address them.

In conclusion, we successfully improved function detection in the SEMA-Toolchain. Furthermore, our approach has demonstrated the potential to improve code coverage by hooking functions more efficiently in the analyzed malware. We

also showed that SAFE can be used to identify common functions within malware families. Finally, we have observed that our binary code similarity detection method could be further optimized by retraining the model with a more diverse dataset.

Bibliography

[1] Annual amount of monetary damage caused by reported cybercrime in the united states from 2001 to 2023. https://www.statista.com/statistics/267132/total-damage-caused-by-by-cybercrime-in-the-us/, Accessed: 11/04/2024.

[2] Ghidra. https://ghidra-sre.org, Accessed: 17/04/2024.

[3] Gonnacry source code. https://github.com/tarcisio-marinho/GonnaCry/tree/master/src/old_version, Accessed: 18/05/2024.

[4] Ida: Lumina. https://hex-rays.com/lumina/, Accessed: 22/05/2024.

[5] Malwarebazaar. https://bazaar.abuse.ch, Accessed: 10/05/2024.

[6] Radare2. https://rada.re/n/index.html, Accessed: 22/04/2024.

[7] tensorflow implementation of word2vec. https://www.tensorflow.org/text/tutorials/word2vec, Accessed: 28/04/2024.

[8] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19:31, 2005.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):1–39, May 2019.

[10] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Khanh Huu The Dam, and Axel Legay. Tool paper-sema: Symbolic execution toolchain for malware analysis. In *International Conference on Risks and Security of Internet and Systems*, pages 62–68. Springer, 2022.

[11] Charles-Henry Bertrand Van Ouytsel and Axel Legay. Malware analysis with symbolic execution and graph kernel. In *Nordic Conference on Secure IT Systems*, pages 292–310. Springer, 2022.

[12] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 169–182, Raleigh North Carolina USA, October 2012. ACM.

[13] Bertrand Van Ouytsel C.H., Crochet C., Dam K.H.T., and Legay A. Sema toolchain. https://github.com/csvl/SEMA-ToolChain, Accessed: 17/04/2024.

[14] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp)*, pages 472–489. IEEE, 2019.

[15] Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity. *ACM Computing Surveys*, 54(3):1–38, April 2022.

[16] Cyprien Janssens de Bisthoven, Zina Rasoamanana, and Ramin Sadre. Openfunctionid: a collaborative database for function identification in software reverse engineering.

[17] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*, pages 338–357. Springer, 2011.

[18] Serena Lucca, Christophe Crochet, Charles-Henry Bertrand Van Ouytsel, and Axel Legay. On exploiting symbolic execution to improve the analysis of rat samples with angr. In *International Symposium on Foundations and Practice of Security*, pages 339–354. Springer, 2023.

[19] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, Hong Kong China, November 2014. ACM.

[20] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. SAFE: Self-Attentive Function Embeddings for Binary Similarity. 2018.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[22] Nikola Milošević. History of malware. 2013.

[23] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, 2017.

[24] James Patrick-Evans, Moritz Dannehl, and Johannes Kinder. XFL: Naming Functions in Binaries with Extreme Multi-label Learning. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2375–2390, San Francisco, CA, USA, May 2023. IEEE.

[25] Cassius Puodzius, Olivier Zendra, Annelie Heuser, and Lamine Noureddine. Accurate and robust malware analysis through similarity of external calls dependency graphs (ecdg). In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–12, 2021.

[26] Najah Ben Said, Fabrizio Biondi, Vesselin Bontchev, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, and Jean Quilbeuf. Detection of mirai by syntactic and behavioral analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 224–235. IEEE, 2018.

[27] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings 2*, pages 35–43. Springer, 2010.

[28] Sebastian Schrittwieser, Patrick Kochberger, Michael Pucher, Caroline Lawitschka, Philip König, and Edgar R. Weippl. Obfuscation-Resilient Semantic Functionality Identification Through Program Simulation. In Hans P. Reiser and Marcel Kyas, editors, *Secure IT Systems*, volume 13700, pages 273–291. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Computer Science.

[29] Paria Shirani, Lingyu Wang, and Mourad Debbabi. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 10327, pages 301–324. Springer International Publishing, Cham, 2017. Series Title: Lecture Notes in Computer Science.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[31] Bastien Wiaux, Arnold Gauthier, and Axel Legay. Building a mutation tool for binaries: expanding a dynamic binary rewriting tool to obfuscate malwares.

[32] 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. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, Dallas Texas USA, October 2017. ACM.

Appendix

repositories

The code used in this thesis is available at `https://github.com/Lix0u/SAFE` and the integration of SAFE in the SEMA-Toolchain as well as the evaluation scripts of the performances of this integration are available at `https://github.com/Lix0u/SEMA-ToolChain`.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**