UCLouvain

École polytechnique de Louvain

Identifying Known Functions to Improve Malware Analysis

Author : **Antoine Mouchet**
Supervisor : **Axel Legay**
Readers : **Charles-Henry Bertrand Van Ouytsel**, **Cristel Pelsser**
Academic year 2023–2024
Master [120] in Cybersecurity

Abstract

Function clone detection is a crucial task in malware analysis as this avoids repeating the complete analysis of a known function. With the ever-evolving landscape of cybersecurity, the increasing complexity of malware outpaces traditional detection techniques. This thesis aims to improve the efficiency of the Symbolic Execution toolchain for Malware Analysis (SEMA) by adding an extended version of SimID as a preprocessing step. SimID is an obfuscation-resilient approach based on the observation of input-output pairs to detect known functions. By shifting from a pattern-matching to a semantic signature-based methodology, we intend to improve the robustness of the toolchain in the presence of obfuscation. We exhibit the added value of our extended version of SimID by showing diverse applications of its different modes. We demonstrate the detection of functions in different compilation and obfuscation contexts with a unique known signature which is an improvement on the current approach. We perform this demonstration in toy binaries, complex programs, and real-world malware. Furthermore, we explore the use of our tool in combination with manual analysis in the case of a lack of resources. Finally, we suggest future research directions to address the current limitations of our approach and to improve the effectiveness of both our version of SimID and the SEMA toolchain more generally.

Acknowledgements

First of all, I would like to express my gratitude to everyone who supported me during the writing of this master thesis.

Primarily, I would like to thank my supervisor Prof. Alex Legay for allowing me to work on this exciting subject and for his consistent cheering during the year.

I would like to express my utmost gratitude to Charles-Henry Bertrand Van Ouytsel for his guidance, feedback, and support throughout the year. More generally, I would like to thank the whole group of students and researchers associated with Pr. Legay for their insights all year long.

Furthermore, I would like to thank the readers and members of the jury, Prof. Axel Legay, Charles-Henry Bertrand Van Ouytsel, and Prof. Cristel Pelsser for taking the time to review this work.

On a more personal note, I would like to thank my family and friends for their unwavering support over the last few months. Thank you for reassuring me in moments of doubt and pushing me to do my best. Finally, I thank my computer for sticking with me the whole year. It would have been way harder without it.

Contents

Chapter 1

Introduction

In modern times, with companies fastening digital transformation and individuals using the Internet on a day-to-day basis, cybercrime is growing bigger and bigger. The cost of cybercrime worldwide is expected to reach 10.5 trillion US dollars in 2025 according to Cybercrime Magazine [1].

Since the early days of malware, those have constantly evolved. Going from simple programs intended as jokes to complex binaries used for nefarious purposes. Therefore, the need for resilient approaches is thriving as traditional malware detection techniques often struggle with obfuscation methods. Obfuscation consists in altering the binary to hinder its analysis. Malware authors rely on various techniques such as instruction substitution, bogus control flow, or virtualisation. Each has its particularities but they are efficient in preventing classical static analysis.

This work presents the addition of a preprocessing step utilising SimID, an obfuscation-resilient functionality detection approach [2] to the Symbolic Execution for Malware Analysis (SEMA) toolchain [3]. This amelioration aims to reduce the impact of obfuscation on the toolchain. Currently, the toolchain uses a pattern-matching approach to detect and replace previously encountered problematic code segments. However, this approach suffers from the fact that the pattern often changes between two different mutants of the same malware. By transitioning from a pattern-matching to a signature-based methodology, this work wants to provide a more robust solution for identifying known functions.

The SEMA toolchain leverages the angr framework for symbolic execution. It generates a System Call Dependency Graph (SCDG) used to classify the binary with a Machine Learning (ML) algorithm. It is capable of classifying malware with high accuracy [3]. However, it still suffers from the shortcomings of symbolic execution and some functions need to be hooked to improve performance [4]. The current approach relies on a pattern-matching algorithm which can be easily

countered with simple obfuscation techniques.

In this work, we extend SimID to use it as a modular functionality detection tool. The detected functionalities can then be hooked before the construction of the SCDG in the SEMA toolchain which alleviates the workload on the symbolic execution. We will show that our approach is efficient even in the presence of obfuscated malware.

This work is organised around three main chapters. Chapter 2 provides the theoretical framework. It discusses static and dynamic analysis, symbolic execution, and syntactic and semantic features in Section 2.1. It continues with function detection approaches in Section 2.2 before presenting the SEMA toolchain in Section 2.3. The chapter closes with the presentation of relevant malware and functions for this work in Section 2.4. Chapter 3 details the contributions of this work, including the extensions SimID in Section 3.1, the signature database in Section 3.2, and the preprocessing step with the reasoning behind it, its implementation, and its advantages in Section 3.3. Then, Chapter 4 presents experimental results, showcasing the obfuscation-resilient capability of SimID in the presence of both dummy and real malware. Chapter 5 outlines potential sources of improvement and thoughts. Finally, Chapter 6 concludes this work wrapping everything up.

Chapter 2

Theoretical Framework

In this chapter, we will introduce all the necessary concepts to explain the work described in Chapter 3. We will talk about the different malware analysis techniques and their specific features in section 2.1, among which we will mention semantic features, which are an essential part of this work. Then, we will delve into function detection and the different approaches to tackle it in section 2.2. We will continue with a presentation of the SEMA toolchain in section 2.3, focusing on the angr component. Finally, we will test our amelioration using different samples. Therefore, we will describe some of their characteristics and the reasons behind their selection in section 2.4.

2.1 Malware Analysis Techniques

Malware analysis aims to derive the maximum amount of information from a binary to determine if it is malicious or benign [5]. There are numerous ways to analyse malware [6, 7]. In this section, we will first discuss static and dynamic analysis with their limits. We will continue with symbolic execution and its drawback. Finally, we will shortly describe syntactic- and semantic-based features in the context of binary analysis.

2.1.1 Static Analysis

Static analysis studies a binary without executing it. We distinguish different versions of static analysis. The simplest technique is information extraction. It involves extracting syntactic features (see section 2.1.4 for details) from the binary to generate a signature. Another way to gather information for this signature is to disassemble the binary. It means that the binary is turned into a low-level assembly version of itself. In this state, more information about the code can be

Bytecode Instructions	Assembly language	Decompiled C code

`31 c0`
`b8 05 00 00 00`
`bb 0a 00 00 00`
`01 d8`
	`xor eax, eax`
	`mov eax, 0x5`
	`mov ebx, 0xA`
	`add eax, ebx`
		`int a = 0;`
		`a = 5;`
		`int b = 10;`
		`a = a + b;`

Table 2.1: Different possible representations of the same code.[3]

retrieved to improve the signature. This syntactical signature can then be used by diverse tools to check it against the ones of known malware to detect if a match is found [5, 7, 8]. The more complex technique is decompilation. With this approach, the analyst turns the binary into a high-level human-readable code representation using a tool such as Ghidra [1] or Hex-Rays [2]. Table 2.1 displays three possible representations of the same binary. With decompiled code, an analyst can investigate a binary more in-depth. However, most malware are obfuscated which complicates the understanding of the code.

2.1.2 Dynamic Analysis

Dynamic analysis intends to solve the problem caused by malware obfuscation. It does not rely on the information within the binary but on its behaviour. Dynamic analysis techniques execute the malware in a controlled environment while monitoring its behaviour at different levels. There are a lot of things that can be monitored among which we can find the following: memory accesses, running processes, opened ports, Windows registries, network & execution flows, and function calls. After execution, the approach will generate a behavioural signature based on the observations made during the execution. Then, similarly to static analysis, the signature is checked against the ones of known malware to see if there is a match.

An analyst can perform dynamic analysis with different tools such as a simple debugger, or a fully dedicated sandbox such as Cuckoo[4]

The inconveniences of approaches using dynamic analysis are multiple. First, it is complicated to provide an analysis system with internet access capabilities

[1] https://ghidra-sre.org/

[2] https://hex-rays.com/decompiler/

[3] For the sake of clarity, we limited our example to a few instructions. In a real-world example, all 3 versions would be bigger.

[4] https://cuckoo.readthedocs.io/

because we cannot let the virus run free. However, it can prevent the malware from showing every capability. This kind of behaviour is called anti-sandboxing. It is a type of evasion technique used by malware to avoid analysis when run inside a VM. When noticing it is inside a sandbox, the malware will only exhibit benign behaviour [5, 6, 7, 8]. In this case, the analysis only considers a single path. This leads us to the last analysis technique.

2.1.3 Symbolic Execution

Symbolic execution is a powerful program analysis technique that abstracts program inputs as symbolic variables instead of concrete values. It allows the exploration of multiple execution paths simultaneously. Symbolic execution relies on a symbolic execution engine and a model checker. The symbolic execution engine keeps track of each path list of constraints and its mapping of variables to symbolic expression or value. The model checker verifies if each path is realisable (i.e. if a concrete value can satisfy all constraints) to prune the unrealisable ones [7, 9].

However, this approach can be hindered by opaque predicates (see 2.2.2 or JIT code (see 2.2.2). It also suffers from the path explosion problem. Indeed, at each conditional jump, 2 branches will be created by the engine. Therefore, it increases the workload of the analysis. For large binaries, this can cause the symbolic execution to fail as memory is exhausted. There are some techniques to mitigate this problem such as search-strategies but it does not resolve it completely [7, 9].

Angr is a tool using symbolic execution that we will use in our approach, we will describe it later in this work. We can now focus on the differences between syntactic and semantic features linked to their concepts of similarity.

2.1.4 Syntactic and Semantic Features & Similarity

Syntactic features are the available ones without executing the malware. They are intrinsic pieces of information about the virus. They can include its length, entropy, file type (magic number), PE headers information, and strings. Execution flows have an ambiguous position. On the one hand, when an analysis uses a statically generated Control Flow Graph (CFG) for detection, it is a syntactic feature. In this case, it only represents a structure and not the information conveyed by it. The problem of such features is that they are easy to alter which can hinder an analysis using them. On the other hand, if it observes the actual execution paths during runtime, it's a semantic feature.

Semantic features on the other hand represent a specific behaviour. An analyst can obtain them by executing or simulating the binary. Function calls, input-

output pairs of individual functions, network connections, and side-channel usage are all examples of semantic features of malware. They are often more resistant to obfuscation techniques as mutating a functionality is harder than modifying the instructions to reach it.

This leads us to define *Syntactic Similarity* and *Semantic Similarity* [2]. The first one can be defined as the comparison of direct code where the inner properties have to match. Whereas, the latter searches for an equivalent behaviour. Therefore, it does not take into account the code internals and is less sensitive to obfuscation. We will now define the concept of function detection and its different methods with regard to the differences we just explained.

2.2 Function Detection

The concept of function detection can be related to the one of binary clone detection and has a wide number of application fields ranging from vulnerability discovery to malware detection including plagiarism detection as well. All those fields share the same interest in finding re-used code and face similar challenges [10, 11] that we will describe a bit further in this writing.

The first approaches date back to 1999, they focused only on the syntactical aspects of the binaries and used either sequences of instructions (EXEDIFF [12]) or functions and basic blocks (BMAT [13]) as points of comparison. In the next decade, some works started taking the semantics into account and applying this research to malware. In [14], they detect polymorphic variants using a graph colouring technique. BinHunt [15] was the first tool to use symbolic execution to check whether two blocks implement the same functionality [11]. More recently, the focus shifted to machine learning approaches with embedding techniques and cross-architecture detection [10, 16].

As seen before, there are numerous levels of granularity at which we can work to identify a clone. From here, we will only focus on techniques looking for functions as units of comparison. Indeed, the goal of this thesis is to identify known functions. Therefore, we will not expand on techniques having a coarser approach comparison level than the function one. The different levels of analysis are illustrated in Figure 2.1.

However, as noted in [10], the actual granularity of the pieces analysed can vary as long as it is finer than the one of the approach. Therefore, we will expand on techniques using functions, blocks, or instructions because they are the ones meaningful in the context of function detection.

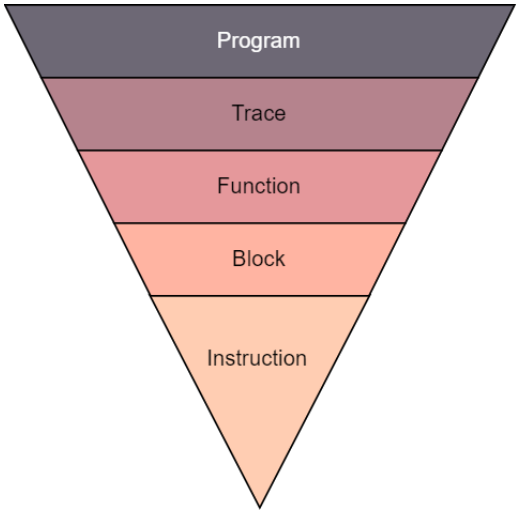


Figure 2.1: The levels of granularity at which comparison can be applied at and within a binary

Speaking strictly about function detection approaches, their goal can be stated as relating a previously unknown function to one already seen before [16]. The comparison result for binary code can be either identical, similar, or equivalent. To be identical, both the source and binary representation have to match. If similarity is the objective, then the source codes of both functions have to be the same even though the binary representation is not. This situation often happens due to compiler optimisations. Finally, the result we are interested in is equivalence because the representations do not matter. As long as the functions have the same semantics, it should be considered a clone [10].

We will now review some recent methods to detect binary function clones as well as the problem caused by obfuscation in this context. We will start with machine learning and function embedding techniques and their limits. We will continue with the challenge of obfuscation in malware and the importance of semantics in this context. Finally, we will focus on the input-output-based approaches, which constitute one of the foundations of this work by diving into SimID [2] and some of its predecessors.

2.2.1 Machine Learning and Function Embeddings

Lately, there has been a surge in approaches making use of different machine-learning techniques [10, 16]. This is due to the fact that those algorithms can process a large number of ground-truth data available for training. Among those approaches, the ones trying to generate function embeddings based on the fea-

Name	Description
Raw bytes	Bytes of the binary function.
Assembly code	Instructions of the binary functions obtained by a disassembler.
Normalized assembly code	Same as above + abstraction to remove some of the variability of operations/operands.
Intermediate Representation (IR)	Lifting of the binary representation to the IR. It keeps the semantics while abstracting from a specific architecture.
Structure	Graphs of internal structure. It is often used in combination with some other representation.
Data flow analysis	Data-flow dependencies, data flow edges used to capture a function's behaviour.
Dynamic analysis	Extraction of features based on dynamic analysis techniques.
Symbolic execution and analysis	Symbolic execution to analyze all paths.

Table 2.2: Different representations to build embeddings on, by increasing level of abstraction [16]

tures of the binary are the most pertinent for our task of known function detection. Some other uses of ML include clustering similar pieces of binary (malware family attribution) or classification of binary code compiled from the same source (plagiarism/copyright infringement) [10].

An embedding is a set of features, also called a vector, summarising information about the function it represents. Those vectors are automatically generated by the different models in the approaches interesting to our goal. They use numbers to represent features, this reduces the dimensionality which helps ML models. The representation on which embeddings are built can vary as can be seen in Table 2.2. An embedding can combine information from different levels of abstraction to better represent a function and its context [16].

Asm2Vec [17] uses a neural-network-based representation learning model to detect semantically similar functions. It is the first approach to combine static features with the lexical semantics relations between tokens (i.e. understanding what

a function actually does) in assembly code. For example, according to Asm2Vec, *memcpy* and *strcpy* are similar to each other. The features vector of this embedding contains information about both the assembly code and the structure in the form of a Control Flow Graph (CFG). This combination captures the context of instructions which provides robustness against common obfuscation techniques. However, it suffers from multiple drawbacks. Firstly, its performance declines when multiple obfuscation techniques are applied concurrently [18]. Secondly, it is not applicable for cross-architecture clone detection as it is trained on a single assembly language [16, 17].

Another approach, PalmTree [19] pre-trains automatically an assembly language model for instruction embeddings through self-supervised learning. PalmTree is not a tool in itself but it generates function embeddings that can be used for downstream applications such as function clone detection ones. The main contribution of this approach is its basis on a BERT model [20] with three training tasks. First, the Masked Language Model (MLM) consists of drilling the model to predict hidden or erroneous tokens within a given instruction. Then, Context Window Prediction (CWP) teaches the model to recognise co-occurring instructions. Finally, Def-Use Prediction (DUP) aims to add data dependency information to the model. We can then use some kind of pooling on the output hidden layers to use them to generate instruction embedding. The downside of PalmTree is that it is computationally expensive to train on fine-tuned data. Moreover, it is not designed to learn a language for cross-architecture work. Therefore, the same function in two different architectures might go unnoticed [19].

A third approach is SAFE [21]. Self-Attentive Function Embeddings (SAFE) wields a Self-Attentive Neural Network to identify similar functions. SAFE functions in two steps. The first one is the transformation of a sequence of assembly instructions belonging to a function into a sequence of vectors (instruction2vec). This model is trained with a skip-gram method which allows the model to include predictions about the instructions around the analysed one. The vectors are then fed to the neural network to output a single function embedding. It combines both the information within the vector and its context in the function. In comparison to Asm2Vec [17], SAFE does not include the CFG inside its embedding, making it more lightweight. Furthermore, it is designed for a cross-architecture approach. However, SAFE suffers in the presence of obfuscation methods [18, 22] because of the nature of the features used for its embedding.

The last approach is DEXTER [23], a function embedding relating semantics to labels for naming functions in binaries. Even if binary function detection is not the main objective of this embedding, it brings interesting development to the table. Indeed, DEXTER combines deep learning, static analysis features (function

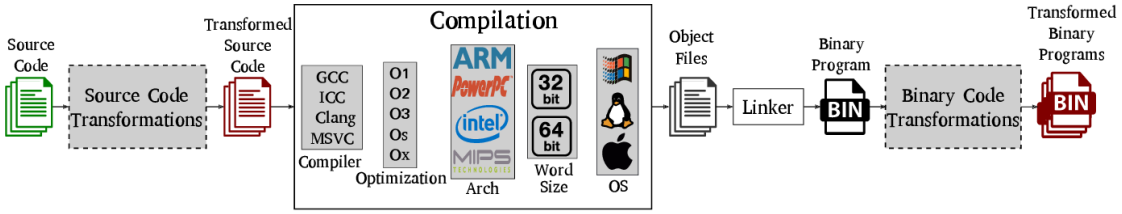


Figure 2.2: Extended compilation process. Greyed boxes are steps at which alterations can occur. Dotted boxes are optional steps where malware can apply obfuscation. Figure from [10]

features vector), context from the call graph (function context vector), and global context (binary context vector). The final embedding of DEXTER is built through an auto-encoder with those three intermediate vectors as input. The function features vector contains quantitative features representing the function lifted to its IR to abstract from a given architecture. DEXTER performs better than all the previously cited embedding due to its manual feature engineering which includes code semantics as expected as shown by PalmTree [19]. Despite being the best, it still suffers from obfuscation methods which prevent disassembly [23].

In conclusion, all those approaches suffer from common shortcomings in our task of identifying known functions in malware. The first limitation is that most of the training data come from cleanware whereas we would like to apply those techniques to malware. Building a dataset of malware source code is intrinsically difficult because authors of such executables do not want it to leak otherwise it would be easier for antivirus companies and enterprises to counter them. Therefore, the ground truth for this kind of binary is very restrained. It relies on the works of reverse engineers which is limited and does not even constitute a ground-truth in itself as each professional will have its own experience. Finally, as seen in the different approaches, obfuscation is a real problem for most of them because the embeddings often rely on static features such as assembly instructions, which can be altered by obfuscation. We will detail these in the next section on the obfuscation techniques used by malware and the challenges it causes.

2.2.2 Obfuscation and Evasion Techniques in Malware

To hinder analysis, malware uses diverse anti-analysis and obfuscation techniques. Some of them even carry the capability to crash the disassembly, or to provide the wrong control flow graph [24]. However, malware is not the only thing affecting a binary. Indeed, the compilation process can also introduce differences due to compiler, optimisations, architecture, word size, and operating system. Thus, the concept of obfuscation in malware is tied to the one of compilation. Malware

authors can alter source code before building their binaries, during the compilation process, or even after it. The extended compilation process with each step able to apply a modification to the final binary is shown in Figure 2.2 from [10].

Before diving into the different obfuscation techniques, we will describe the evolution of the types of obfuscated malware.

In 1988, the CASCADE virus [25] was the first one to *encrypt* its body. When launched, the decryption routine would start before passing the execution to the freshly decrypted body. As the virus propagated, the encryption algorithm came to light. It led to the creation of a new kind of malware whose decryption would be mutated to avoid detection by emulation analysis: *oligomorphic* malware. This kind of virus obfuscates its decryption routine to prevent signature-based detection [5, 8, 24, 26], it was the case of the Whale DOS virus in 1990. The problem with this type was that the number of decryptors is bounded. Therefore, an anti-virus can exhaustively list all possibilities. *Polymorphic* malware was the next step to solve this problem for malware authors. It did not only obfuscate the decryption routine but its content as well. This removed the problem of having a finite number of decryption algorithms. A polymorphic virus, such as V2PX in 1990, encrypted the content of the decryption algorithm (which was chosen randomly for each infection) [5, 8, 24, 26]. However, the limitation stays the same as the body of the virus does not change and can be detected when decrypted. This is where the last type of obfuscated malware, *metamorphic* malware, came into play. They introduced the concept of code obfuscation so that no two viruses, even from the same family or source code, have the same signature. The first sample to use this technique appeared in 1998, W95/Regswap [5, 8, 24, 26]. Despite being more than 20 years old, this kind of malware is still present due to the evolution of the obfuscation and anti-analysis techniques it uses. Figure B.1 in Appendix B from [24] provides a detailed timeline from 1971 to 2017 of malware and obfuscation techniques evolution.

The name "anti-analysis techniques" is self-explanatory but they can be divided into different sub-groups: anti-file processing such as Chameleon or Werewolf attack [5], anti-debugging [5], and the most common anti-Virtual machine [5, 7, 8]. The last ones try to detect the presence of the virtual machine environment through diverse means such as user activity checking [7], or sandbox fingerprinting [8]. A more complete list of evasion techniques can be found at https://evasions.checkpoint.com, the Check Point Research encyclopedia about evasion techniques [27].

Concerning obfuscation techniques, they have evolved since their inception. We will mention them by increasing complexity level and we will present some examples. Some of them are used by common obfuscators such as Obfuscator-LLVM [28] or Tigress [29] which we will use in our evaluation.

Before obfuscation	Variant 1	Variant 2

`xor eax, eax`
`mov eax, 0x5`
`mov ebx, 0xA`
`add eax, ebx`
	`xor eax, eax`
	`mov eax, 0x5`
	`nop`
	`nop`
	`mov ebx, 0xA`
	`nop`
	`add eax, ebx`
		`xor eax, eax`
		`xor ebx, ebx`
		`mov eax, 0x5`
		`mov ebx, 0xA`
		`xor ecx, ecx`
		`mov ecx, eax`
		`add ecx, ebx`
		`sub ecx, 0xC`
		`add eax, ebx`

Table 2.3: Examples of *Dead Code Insertion* adapted from [24]. Variant 1 uses nop. Variant 2 adds a useless code sequence

Dead Code Injection

Also called *Junk Code Insertion*, this technique adds instructions to the binary without affecting its functionality. The inserted code ranges from operations that do nothing (NOP - No OPeration) to more complex code sequences that do not affect the binary's final behaviour [5, 8, 24, 26]. Table 2.3 presents two examples of this mechanism.

Code Transposition & Subroutine Reordering

Code Transposition and *Subroutine Reordering* (or *Block reordering*) work in a similar fashion. Both intend to shuffle the code so that the order of bytes in binary is different from the order of execution. On the one hand, *Code Transposition* uses either conditional and/or unconditional jump or direct reordering of instructions that have no dependence relationship [5, 24, 26]. Whereas *Subroutine Reordering* randomly permutes the order of the subroutines (or blocks) [8, 24, 26]. The number of potential variants equals $n!$ where n is the number of instructions, blocks, or subroutines depending on the chosen technique. Table 2.4 displays the combination of both techniques.

Register Reassignment

This technique assigns values of registers and memory variables used by the program to the ones that are currently free [5, 8, 24, 26].

Before obfuscation

```
1 mov eax, ecx
2 mov ebx, 0xA
3 mul ebx
4 add eax, 0x5
5 mov ecx, eax
```

Variant

```
1     mov ebx, 0xA
2     jmp F1
3 F2: add eax, 0x5
4     jmp F3
5 F1: mov eax, ecx
6     mul ebx
7     jmp F2
8 F3: mov ecx, eax
```

Table 2.4: Example of combination of *Code Transposition* and *Subroutine Reordering* adapted from [24].

Instruction Substitution

Instruction Substitution in its basic form replaces a binary operation with a semantically equivalent one in order to delay the analysis [5, 8, 24, 26]. However, Obfuscator-LLVM and Tigress take it further by replacing an instruction with a more complex sequence of code. An instruction can be any of the following: literals, data, arithmetic operation, API call, jump target [30, 31]. Table 2.5 presents the 2 approaches side-by-side. Table 2.5a shows a simple example of *Instruction Substitution* and Table 2.5b displays the capacity of the *EncodeArithmetic* feature of Tigress.

Flower Instruction

Flower Instruction is a special kind of code injection as its goal is either to completely block the disassembly process using specially crafted instructions or to mask the control flow with the Windows Exception Handling Mechanism [8].

Subroutine Inlining & Outlining

Whereas *Subroutine Inlining* replaces a call to a function by the code of its body [24, 31], *Subroutine Outlining* creates a function using a block of code and makes an unconditional jump to it [24].

Opaque Predicate Addition

Opaque Predicates are hard to analyse conditional jumps whose condition will always evaluate to True or False at runtime. It creates complex evaluation con-

Before obfuscation	Variant

`xor eax, eax`
`mov eax, 0x5`
`mov ebx, 0xA`
`add eax, ebx`
`mov ecx, eax`
	`xor eax, eax`
	`mov eax, 0x5`
	`mov ebx, 0xA`
	`add eax, ebx`
	`push eax`
	`pop ecx`

(a) Simple *Instruction Substitution* inspired from [24]

Before obfuscation	Variant
`z = x + y + w`	`z = (((x ^ y) + ((x & y) << 1)) | w) +` `(((x ^ y) + ((x & y) << 1)) & w);`

(b) *EncodeArithmetic* example from [32]

Table 2.5: Two Methods of *Instruction Substitution* Obfuscation

straints for static or symbolic analysis, slowing it down considerably [7, 31].

Bogus Control Flow

This technique is specific to the Obfuscator-LLVM project. It modifies the Control Flow Graph of a function by mixing *Opaque Predicates*, *Code Transposition*, and *Junk Code Insertion* together. A basic block filled with ineffective code is added before the function, it terminates with a conditional jump, using an opaque predicate, to the original entry point of the function. The example from [30] is quite explanatory as shown in Figures C.1 and C.2 in Annex C.

Control Flow Flattening

Control Flow Flattening is a technique available in both obfuscators. It totally wipes the structure of the CFG, making it as flat as possible. This prevents analysis techniques based solely on CFG from detecting malicious software using this approach [30, 31].

Argument Randomisation

Tigress implements a specific transformation that randomly reorders the arguments of a function. Moreover, it can add bogus parameters [31]. This obfuscation technique is painful for our method since we rely on the observation of known input-output pairs. If the prototype changes, especially the number of parameters, our approach could suffer a deterioration in performance [2].

Virtualisation

Virtualisation transforms a function into a specialised interpreter with its own byte code language. At runtime, the virus interprets the byte code by looking at the corresponding machine code instruction and executing it [31]. It is particularly efficient to counter static analysis [7].

Just-In-Time (JIT) Code

Malware using *JIT Code* write their next instruction at execution time. It is a very powerful tool to impede symbolic analysis as the engine will not resolve the symbolic constraints. Therefore, it will get stuck on symbolic instructions or addresses [7]. Furthermore, Tigress implements a dynamic version of this approach so that the code is continuously modified [31].

With all those obfuscation techniques, the need for appropriate and resilient approaches is at a peak. According to [10], semantics approaches are in the best position to address those challenges. We will now delve into those approaches, and most specifically, the ones using input-output pairs.

2.2.3 Input-Output Based Approaches

In this section, we focus on approaches using input-output pairs to identify semantically equivalent code. Those approaches are often more resilient to obfuscation because they do not rely on a binary representation to find a match. Moreover, they do not have a training phase and are cross-architecture. This last argument is one of the main advantages of choosing this type of technique to identify known functions in malware. We will now review different tools using this approach.

In 2009, [33] pioneered this field with a method focusing on automatic mining of functionally equivalent code fragments using random testing. The intended application for their work is to improve code understanding and maintenance. Their approach aimed to identify code clones at the instruction level based on functional equivalence rather than syntactic similarity using data-flow analysis.

However, as seen previously in section 2.2.2, this kind of methodology is sensitive to different obfuscation techniques. Therefore, it is not suitable for malware analysis.

In 2012, [34] proposed Aligot which emphasises cryptographic function identification within obfuscated binary programs by using their I/O relationship without consideration for the actual implementation. This approach highlights the inefficiency of previous tools due to the lack of reliable static features in obfuscated code. It uses a specific loop detection and extraction algorithm to capture the exact input and output values of cryptographic routines within traces of dynamic analysis. The loop data flow is then matched against known cryptographic references. It does so by using the extracted input with the reference implementations; if the outputs (the extracted one and the analyzed one) match, then the function is detected. This approach can manage obfuscated cases as long as they do not break the loop architecture. However, it still suffers from the usual dynamic analysis shortcomings. Moreover, Aligot is not able to retrieve the corresponding function if the obfuscation breaks the loop. Finally, it was designed as a proof-of-concept and is not feasible for real-world scenarios where time matters.

CACompare [35] uses randomly generated input and the corresponding output values with comparison operands, their condition, and library function calls to create a semantic signature. To allow for cross-architecture search, this technique abstracts from a specific platform in a similar fashion to what Multi-MH did [36]. Indeed, they lift the binary assembly to an Intermediate Representation before building the signatures. CACompare then uses its known signatures to match each function with its most similar one in the target binary. The main difference with Multi-MH is that CACompare removes the need for the CFG for the search. It uses semantic signatures from function emulations to find matches instead of iterating on basic blocks. However, it is still vulnerable to obfuscation techniques targeting the control flow of a binary because it can heavily alter the comparison operands and conditional code.

More recently, [2] introduced a very promising obfuscation-resilient approach using program simulation into which we will dive directly as it constitutes the basis of this work.

SimID

SimID [2] is an "obfuscation-resilient semantic functionality identification approach" developed by Schrittwieser et al. in 2022. SimID will identify a functionality inside a given binary if any function within this binary matches the expected input-output pair of the functionality without considering the actual implementation or the temporary states of the program. It counters some common obfuscation

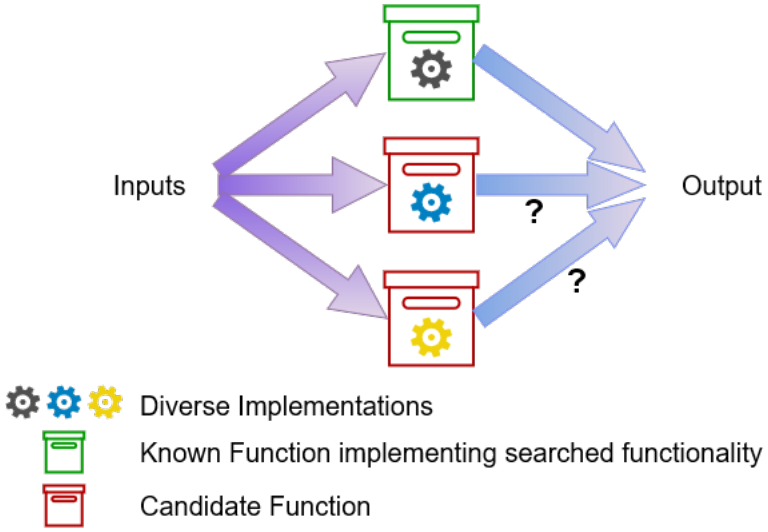


Figure 2.3: General concept of SimID

techniques such as function splitting, bogus control flow, or virtualization. The general concept of SimID can be seen in Figure 2.3

SimID builds on top of its two main interleaved features *Program Simulation* and *Function Input-Output Matching*. Both rely on different angr [37] components. *Program Simulation* can create an abstracted execution environment that mimics real-world execution. It helps symbolic execution by simulating an abstracted version of a function call behaviour. This is particularly convenient when the arguments of a function are complex data structures as is often the case in malware. We will now delve into *Program simulation* and describe the three angr pieces relevant to this part. Then we will dive into the *Function Input-Output Matching*.

The first is *Valgrind's VEX IR* [38]. It is necessary to run an abstracted version of the original binary. The objective of the VEX lifter is to translate the low-level assembly code to a higher-level format, the Intermediate Representation (IR). This lifting allows SimID to abstract from architectures. Indeed, the IR describes the semantic operations executed by the CPU without relying on specific instruction sets.

SimProcedures are the second component. They matter because they are required to represent the calls to external variables or functions. We will provide further detail on these in section 2.3.1 dedicated to angr.

The last one is the *call_state()* constructor, one of the numerous state construc-

[5] Adapted from the angr documentation example [38] and from vex repository [39].

`adds R1, R2, #15`	`t0 = GET:I32(16)`
	`t1 = 0xF:I32`
	`t3 = Add32(t0,t1)`
	`PUT(8) = t3`
	`PUT(68) = 0x42DA5:I32`

Table 2.6: ARM instruction vs VEX IR[5]

tors provided by angr. It recreates a simulated program state for the function to be called in. Therefore, it allows SimID to execute a single function isolated from the rest of the binary. It is one of its core functionalities.

Regarding the *Function Input-Output Matching* component, we can break it down into two parts. Firstly, SimID will execute function prototype recovery using angr's logic to retrieve the size and order of the argument of each function in the binary. In a second time, SimID will attempt to actually detect the functionality in the program by following this strategy:

1. Exclude functions where the prototype does not match the one of the desired functionality.
2. Exclude functions where the prototype extraction process failed.
3. Simulate matching functions. If a result is found, stop the analysis.
4. Simulate functions excluded at points 1 & 2 otherwise.

With this in mind, the pseudo-code of the core algorithm of SimID as expressed in [2] can be found in Algorithm A.2 in Appendix A. However, when checking the source code of SimID, it presents some algorithmic differences with the pseudocode shown in the paper. The actual algorithm can be seen in Algorithm A.1 in Appendix A. The difference lies in the fact that it does not compute a list of wrong function prototypes but processes or skips them directly based on a predefined boolean parameter. Note that if the function prototype does not match, it will automatically skip it. This difference means that the actual implementation does not privilege functions for which prototypes could be retrieved. Therefore, it might lose time on those. Whereas pruning out functions first could make it faster even if the algorithm still analyses functions with unknown prototypes later.

To optimise the analysis time, we can decide to terminate the search when the functionality is identified. However, this may force us to redo the work in certain

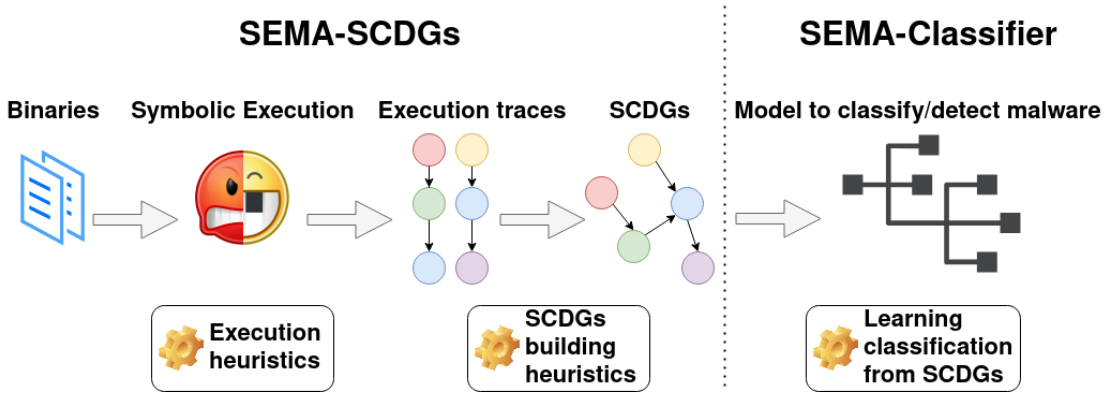


Figure 2.4: Components of the SEMA toolchain from [42]

cases where the first time the feature is detected is a false positive. For this reason, we generally favour letting the algorithm do a full run on the binary and output the list of candidate functions. In our work, we will often stop at the first time the function is found because we estimate that it is sufficient for our task as the number of false positives is quite low in the original paper [2].

We will use SimID to improve the SEMA toolchain by adding a preprocessing step. We will detail our contribution in Chapter 3. Before that, we will describe the SEMA toolchain in the next section.

2.3 SEMA - Toolchain Presentation

SEMA stands for Symbolic Execution toolchain for Malware Analysis. It was proposed in [3]. The need for such a tool arose from the realisation that a large number of current viruses are only polymorphic variants of known malware. It aims to provide a new way to detect malware. The toolchain relies on different components. First, it will ingest a binary and execute it symbolically with angr [40] in order to generate a System Call Dependency Graph (SCDG) by merging all symbolic traces. Finally, it employs various machine-learning algorithms for the detection and classification of malware [41]. Figure 2.4 shows the complete architecture of the toolchain.

SEMA relies on angr for the symbolic execution. Therefore, we will describe this tool in greater detail.

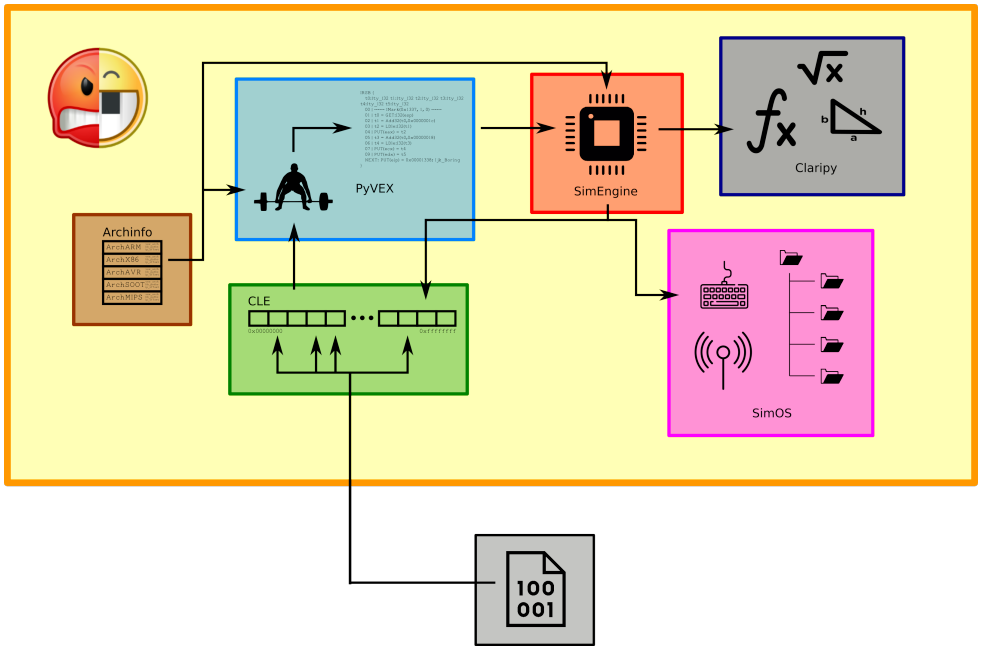


Figure 2.5: Components of angr from [44]

2.3.1 angr

angr [40] is an open-source framework to analyze binaries with an active GitHub community. It offers capabilities for symbolic analysis in both static and dynamic contexts and is used in various fields [43]. It is developed in Python and allows for cross-platform analysis thanks to its inner design which we will develop a bit further down. angr relies on six tools to provide its full evaluation. Figure 2.5 shows the connections between those mechanisms. We will now break down those components following the order in which angr uses each one of them.

CLE

CLE is the loader. It is the first step in the whole angr process. It retrieves the code and tries to understand the program's structure to construct a memory mapping based on it. Its result is a loader object with all the code, the memory, and loaded required libraries.

Archinfo

Archinfo represents the architecture of the binary. It is the object on which other angr components depend the most as it contains all the low-level internal parts

needed to symbolically execute the program. It abstracts from specific platform information so that angr scripts can be fully used cross-architecture.

PyVEX

PyVEX is the module responsible for the lifting of assembly code to the VEX Intermediate Representation (IR). The advantage of the IR is that it is consistent on semantics even if the underlying architectures are different. Therefore, the symbolic execution engine needs to understand only the IR used and not all possible assembly instructions. A translation from machine code to VEX IR is exposed in Table 2.6.

SimEngine

SimEngine is the core of angr, it processes each lifted IR block and updates states and their constraints. It can also create new states when needed (e.g. because it met a conditional branch). In short, it is the symbolic execution engine.

Claripy

If SimEngine is the core of angr, then Claripy is its brain. Indeed, it processes all the symbolic execution and constraints, it can even resolve them when required. It is the model checker part of the symbolic execution.

SimOS

It provides high-level abstractions usually provided by the operating system such as network and file interactions. It prevents angr from dealing with hard work commonly done by the OS. Moreover, it offers *SimProcedures* which are symbolic summaries of system calls and common libraries. One can also extend the existing procedures or create custom ones. We use them to relieve the symbolic execution engine and make angr faster. *SimProcedures* can be hooked to a specific address within the binary, they can replace whole parts of the program. In our work, we will use them to replace the matching functions of a desired functionality. Some procedures are already implemented in angr, they intend to mimic some common API calls. However, we can define our own ones to replace more complex functionalities.

2.3.2 Hooking in SEMA

Coming back to SEMA, it uses different exploration strategies in order to mitigate the path explosion problem to construct the best SCDG possible. However, the

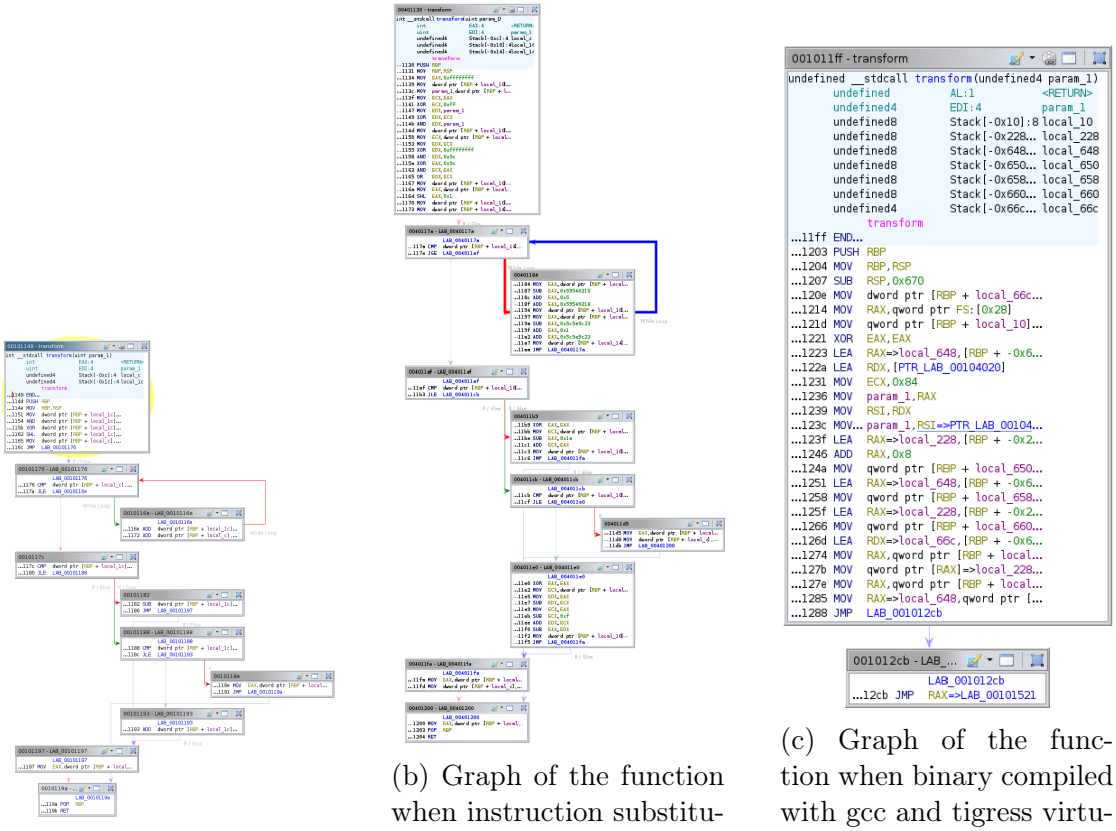
```c
#include <stdio.h>

int transform(int value){
    value &= 0xFF; //=150
    value ^= 0x9C; //=10

    for (int i = 0; i < 5; i++){
        value += 5;
    }

    if (value > 30 ){
        value -= 30;
    }
    else if (value > 0){
        return value;
    }
    else{
       value += 15;
    }
    return value;//=5
}

int main(int argc, char const *argv[]){
    int init_val = 150;
    int result = transform(init_val);

    if (result == 5) {
        printf("I'm evil !");
    } else {
        printf("I'm benign !");
    }
    return 0;
}
```

Listing 2.1: Toy example to showcase the impact of obfuscation

approach still suffers from obfuscation techniques such as virtualisation and the ones manipulating the control flow. Our method aims to reduce those limitations by detecting known functionalities using the technique described in [2] (See Section 2.2.3). When a functionality is identified, it is replaced by a user hook (also called SimProcedure) which will relieve the symbolic execution within the SEMA toolchain.

The current hooking mechanism leverages a pattern-matching approach to hook a functionality[6] in the event that it matches a previously encountered problematic

[6]By functionality, we mean any segment of code implementing different operations.



(a) Graph of the original unaltered function

(b) Graph of the function when instruction substitution from obfuscator-llvm is applied to the binary

(c) Graph of the function when binary compiled with gcc and tigress virtualize applied to the function

Figure 2.6: Three graphs of the same *transform* function with different obfuscation methods

segment of code. The patterns used by this approach consist of series of hexadecimal bytes representing machine instructions extracted from manually analysed functions. However, the results of this technique heavily deteriorate in the presence of obfuscation. Indeed, a predefined pattern might not be found in mutated versions of malware despite the functionality being present. To be clearer, each obfuscation technique modifies the machine instructions of the original binary in a different way as we saw in Section 2.2.2. Therefore, even when the semantic functionality is still present in the binary, a syntactic pattern-matching approach will not detect it and it can still cause problems to the toolchain later on.

Listing 2.1 shows a simple C program computing a constant value and displaying "I'm evil !"[7]. The *transform* function always evaluates to *5* within this binary. We will use this example to depict how obfuscation can hide the pattern associated

[7]Toy malware inspired from [7]

with the original implementation of this function. We will also show later on that the approach described in this work is able to detect it without any problem no matter the obfuscation method used.

Figure 2.6 depicts the graph of each obfuscated version of the function with its assembly code[8]. We clearly notice that diverse obfuscation techniques alter the assembly code in different ways. This observation implies that each one of them escapes detection from a pattern-matching approach. Indeed, the segment code from the unobfuscated sample does not appear in the obfuscated ones. Therefore, they will all evade exposure. We can clearly see that even simple techniques such as instruction substitution are sufficient to avoid detection by a pattern-matching approach.

This thesis aims to improve the functionality detection method by using an input-output approach based on semantics to replace the pattern-matching technique relying on syntactic properties.

2.4 Function Choices & Malware Description

In this section, we will describe various types of interesting functions for our approach. Then, we will justify our choices of malware selection.

First, it is important to note that our approach relies on the input/output relation of functionalities. Therefore, our focus will only be on deterministic functions. By deterministic, we mean that the relation between the input and the output should not be impacted by any randomness or user-specific interactions. For example, cryptographic algorithms often use random generators to avoid reusing the same keys. Therefore, we cannot define a functionality in our database to detect key generation methods. However, decryption has to be deterministic which makes it a good candidate. Hashing algorithms are, by definition, deterministic. Therefore such functions are ideal for our tool. We will now review some functions and the reasons behind our choices in greater detail.

We will start by focusing on domain generation algorithms. DGAs have been broadly adopted by botnets and other malware to hide the actual communication channel among a large list of candidates [45]. Usually functions implementing DGAs take a seed as input and output a string (or a list of strings) representing domain(s) to contact. The relation between both is deterministic which makes DGAs suitable for detection by our tool [2]. As a benchmark, we will use the same dummy malware inspired by the Ramdo malware as the authors of the original version of SimID, the *generate_ domain* function that we will try to detect can be

[8]Another graph with flattening can be found in Appendix D.1

seen in Listing 2.2 (taken from [2]).

```
1  struct node {
2      void *data;
3      struct node *next;
4  };
5
6  typedef struct node * llist;
7
8  struct sSelf {
9      long int seed;
10     long int nr;
11     long int generateddomains;
12     char lastdomain[50];
13     llist *domainhistory;
14 };
15
16 char *generate_domain(struct sSelf *self){
17     long int s = ((2 * self->seed) * (self->nr + 1));
18     long int r = ((long int) s ^ (long int) ((26 * self->seed)
           * self->nr));
19     char domain[50] = "";
20     for (int i = 0; i < 16; i++){
21         r = (r & 4294967295);
22         strcat(domain, chr(((r += ((long int) r ^ (long int) ((
               s * pow(i,2)) * 26));
23     }
24     strcat(domain, ".org");
25     strcpy(self->lastdomain, domain);
26     self->nr += 1;
27     return self->lastdomain;
28 }
```

Listing 2.2: Reimplementation of the DGA of Ramdo

On the other hand, authors in [4] specifically focus on the use case of the SEMA toolchain. Therefore, we will now develop the malware and functions described in this paper. Focus is made on RAT as symbolically analysing them allows an analyst to explore most paths uncovering a variety of actions. One problem experienced by the authors is a non-resilient pattern-matching approach. Indeed, in the presence of syntactic obfuscation, patterns of previously known functions change and are not hooked. In such an event, they need to redo the work. Our approach aims to go from a pattern-matching approach to a signature-based one. The paper lists different problematic functions such as *copy_mem* or *MurmurHash*, a non-cryptographic hash function. Those functions are part of the Warzone Remote Access Trojan (RAT). Therefore, we will show their detection rate using our approach later on.

```
1  int copy_mem(int dest_addr, undefined* src_addr, int length ) {
2      int offset ;
3      if ( length != 0) {
4          offset = dest_addr - (int) src_addr;
5          do {
6              src_addr[offset] = *src_addr;
7              src_addr = src_addr + 1;
8              length = length - 1;
9          } while ( length != 0);
10     }
11     return dest_addr;
12 }
```

Listing 2.3: Function managing memory

Let's finally dive into functions that are usually problematic for symbolic execution. Symbolic execution is sensitive to the path explosion problem. Prevailing contributors to this problem are loop-like structure, exceptions, pointer aliasing, and concurrency [9, 46]. Therefore, functions that have such structures are interesting to detect as they are often time-consuming. The *copy_mem* function in listing 2.3 described in [4] is a perfect example of a function with a loop-like structure.

Regarding the interesting malware, as shown in the aforementioned papers, RATs are among the most suitable malware to use our tool on because they combine multiple of the previously mentioned functionalities. Remote Access Trojans intend to remotely control an infected device. They can spy on users, alter information, etc. An attacker can accomplish nearly any objective because they have complete control over the compromised computer and its operations [47]. The Warzone RAT is quite recent with a lot of evasion techniques which makes it a good use case [48]. We will also use GonnaCry which is an older ransomware but its source code is available online [9]. Therefore, an interesting use case will be to detect some utils functionalities in different obfuscation contexts.

[9] Source code available here

Chapter 3

Contribution

In [4], the authors describe the need for detecting resource-intensive functions in order to reduce the time taken by the toolchain. In this chapter, we describe our contribution which is articulated around three axes. First, we extended SimID and turned it into a modular tool allowing analysts to use it in more various contexts in Section 3.1. Then, we designed a functionality signature database to combine with the modularity of SimID. Moreover, we developed a methodology to create new functionality signatures, both are described in Section 3.2. Finally, we integrate this tool as a preprocessing step of the SEMA toolchain and compare the performance on different samples in Section 3.3.

3.1 New Version of SimID

Originally, the script of SimID had a lot of hard-coded information. It made it hardly extensible which is a problem when trying to detect diverse functionalities regarding various parameters such as number of inputs, types of inputs, and type of output. We need this modularity because the functionalities we will be looking for are not consistent in those aspects. If we had to change the script's inner parts on each iteration, it would rapidly become time-consuming and inefficient. Therefore, we devised three modes that can be used in different use cases. A common feature of all modes lies in the creation of permutation of parameters of searched functionalities. It allows our new version of SimID to partially counter the argument randomisation obfuscation method described in Section 2.2.2. We designed the *generateInput* function to create the concrete inputs to use with angr Callables. It has a verification step to confirm potential matches among functionalities for the current function if the return type could be extracted during prototype recovery, and generates all the concrete values with all permutations for those matches.

Algorithm 3.1: Look for a given functionality within the binary

Input: P (Program in binary representation)
Input: FUNCNAME (NAME of FUNCtionality)
Input: ARGNUM (NUMber of ARGuments of functionality)
Input: ARGTYPES (TYPES of ARGuments of functionality)
Input: DB_FILE_PATH (PATH of the DataBase FILE)
Output: CF (Dictionary of Functions implementing the searched functionality)

```
1  DATA ← readFile(DB_FILE_PATH)
2  CFG ← reconstructControlFlowGraph(P)
3  EOUT, EPROTO, HOOK, FUNCDATA ← accessDB(DATA, ARGNUM,
     ARGTYPE, FUNCNAME)
4  EIN ← generateInput(FUNCDATA, FUNCNAME)
5  for function in P as FUNC do
6  |  CC ← reconstructCallingConvention(FUNC, CFG)
7  |  if CC.PROTOTYPE = EPROTO ∨ (analyze_if_unkwnow_prototype
   |   ∧ CC.PROTOTYPE = None) then
8  |  |  for IN in EIN do
9  |  |  |  CS ← generateCallState(FUNC, IN, EPROTO)
10 |  |  |  C ← createCallable(FUNC, EPROTO, CS)
11 |  |  |  RET ← C(IN)
12 |  |  |  if RET = EOUT then
13 |  |  |  |  CF[FUNC.ADDR] = [FUNC.NAME, FUNCNAME, HOOK]
14 |  |  |  |  return CF
15 |  else
   |  |  // Go directly to next iteration
16 |  |  continue
17 return CF
```

The first mode is the most basic one, it follows the same concept as the original version of SimID. It looks for one functionality inside the binary. We can distinctly see the similarities with its pseudocode in Algorithm 3.1 when comparing it to the pseudocode of the original SimID in Algorithm A.1. The motivation for this approach is to quickly analyze a binary based on previous knowledge. Indeed, if we already suspect the sample to have some functionalities, this mode is the best one to confirm its presence. It will simulate each function within the binary

Algorithm 3.2: Search for all functionalities in the binary

Input: P (Program in binary representation)
Input: DB_FILE_PATH (PATH of the DataBase FILE)
Output: CF (Dictionary of functions implementing a known functionality)

```
1  DATA ← readFile(DB_FILE_PATH)
2  CFG ← reconstructControlFlowGraph(P)
3  for function in P as FUNC do
4      CC ← reconstructCallingConvention(FUNC, CFG)
5      C_DATA, C_PARAM_TYPE ← candidatesGeneration(DATA,
         CONTROL, CC.ARGS)
6      EIN ← generateInput(C_DATA, CC.RETURN_TYPE)
7      i = 0
8      for FUNCNAME, FUNC_IN_VAL in EIN.items() do
9          EOUT, EPROTO, HOOK ← accessDB(DATA,
             len(FUNC_IN_VAL), C_PARAM_TYPE[i] , FUNCNAME)
10         CS ← generateCallState(FUNC, FUNC_IN_VAL, EPROTO)
11         C ← createCallable(FUNC, EPROTO, CS)
12         RET ← C(FUNC_IN_VAL)
13         if RET = EOUT then
14             CF[FUNC.ADDR] = [FUNC.NAME, FUNCNAME, HOOK]
15             break
16     i ← i + 1
17 return CF
```

with the input associated with the functionality in the signature database. If the output matches the expected result, the script writes the address of the matching function, the functionality matching the function, the length of the function, and the name of the corresponding hook to a JSON file. The analyst can then launch the SEMA toolchain with this file to directly hook the detected functions within the sample to the appropriate SimProcedures.

The second mode works in a similar fashion to the first one as shown in Algorithm 3.2 except that it tries to match each function against all known functionalities of the database. This approach can be useful to detect unsuspected components of a binary. The analysis with this mode is obviously slower as the number of simulations grows with both the number of functions present in the binary and the number of functionalities saved inside the database. It constitutes

Algorithm 3.3: Look for a functionality matching a given function in the binary

Input: P (Program in binary representation)
Input: FUNC_NAME (NAME of the FUNCtion in the binary)
Input: DB_FILE_PATH (PATH of the DataBase FILE)
Output: CF (Dictionary of function implementing a known functionality)

```
1  DATA ← readFile(DB_FILE_PATH)
2  CFG ← reconstructControlFlowGraph(P)
3  FUNC ← extractFunctionFromCFGKnowledgeBase(FUNC_NAME)
4  CC ← reconstructCallingConvention(FUNC, CFG)
5  C_DATA, C_PARAM_TYPE ← candidatesGeneration(DATA,
     CONTROL, CC.ARGS)
6  EIN ← generateInput(C_DATA, CC.RETURN_TYPE)
7  i = 0
8  for FUNCNAME, FUNC_IN_ VAL in EIN.items() do
9  |   EOUT, EPROTO, HOOK ← accessDB(DATA, len(FUNC_IN_VAL),
   |     C_PARAM_TYPE[i] , FUNCNAME)
10 |   CS ← generateCallState(FUNC, FUNC_IN_VAL, EPROTO)
11 |   C ← createCallable(FUNC, EPROTO, CS)
12 |   RET ← C(FUNC_IN_VAL)
13 |   if RET = EOUT then
14 |   |   CF[FUNC.ADDR] = [FUNC.NAME, FUNCNAME, HOOK]
15 |   |   break
16 |   i ← i + 1
17 return CF
```

an interesting approach in the case where an analyst does not have any hint as to what kind of binary is currently assessed. The main difference in implementation with the first mode (except the obvious loop for all functionalities) lies in the presence of a new function *candidatesGeneration*. This method, based on the previously gathered knowledge of the current analysed function, creates a curated listing of potential candidates. It makes use of the prototype if available to create a reduced set of candidates. The function is conscious of the prototype availability via the *CONTROL* parameter which is updated during the prototype recovery process. More clearly, it generates candidates for which the prototype matches if it was recovered, or all functionalities of the database if it was not.

Finally, the third mode is intended for a live use with the SEMA toolchain.

Indeed, it uses SimID on a specific function specified with its name within the binary. It might be useful if an analyst notices that one function is blocking the symbolic execution. At this point, the objective would be to divert execution to SimID to determine the functionality behind the time-consuming code section. If our approach finds a functionality, it could tell the SEMA toolchain how to hook it. It would greatly reduce the workload on the SCDG component of the toolchain. Concretely, in this mode, our tool will start by trying to recover the prototype of the given function. On success, it will only check for functionalities matching this prototype in a similar fashion to what happened in the second mode. Similarly, if the analysis is unable to recover the prototype, it will try to match the function with every known functionality. Algorithm 3.3 shows the pseudo-code for this version. The parallels with the second form are easily noticeable and normal since they follow the same idea to generate potential candidates.

3.2 Signature Database and Methodology

The motivation for a signature database arose from the observation that our approach could identify different functionalities. However, modifying the script internals on each iteration was tiresome and time-consuming. The signature database allows us to store as many functionalities with their particularities as we want. Moreover, as explained previously, an analyst does not always know which functionality will be part of a binary. The database enables the possibility to batch search the binary with all its elements at once without modifying the code between two searches.

Using a JSON file to store the database made a lot of sense as it allows for a fast look-up thanks to the onion-layered approach. As can be seen in Figure 3.1, the outermost layer is the number of parameters, then comes the types of those inputs, and finally each functionality with its internal details. Furthermore, a JSON file is easy to understand and scale-up. It is also pretty lightweight compared to a database and can be more easily accessed. Another advantage is that it allows for potential collaboration. Analysts could share their input/output patterns with the rest of the relevant information to create a bigger database which could have multiple uses with our approach being one of them. Another use would be analysts trying to reverse engineer binaries and using this database to help them get an idea of what a function is based on its prototype.

Regarding the structure of the database, it consists of a JSON file structured around the parameters of the functionalities. The first key is the number of input parameters of a function. The number of input parameters can sometimes be retrieved from the function when analyzing it with angr. In the case of such an event, the number of potential functionalities matching this function is reduced.

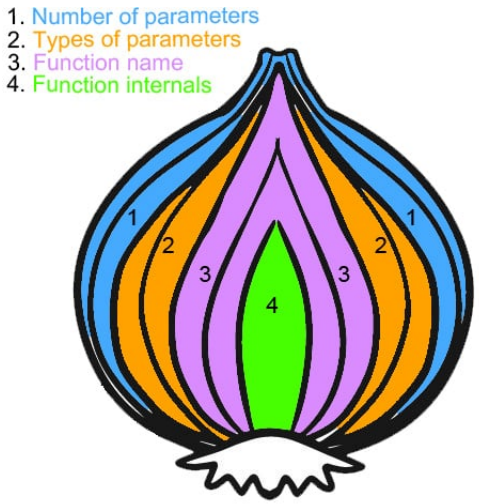


Figure 3.1: Onion-layered view of the JSON file

Indeed, the script will only consider functionalities with the same number of input parameters. In the second position, we find the type(s) of parameter(s). When there are multiple parameters, the order is arbitrary but it does not influence the analysis because, as explained previously, our version of SimID generates all possible permutations for a set of inputs. The third key is the name of the functionality. The last holds various information used during the analysis. Table 3.1 shows the different attributes of each functionality.

We will now describe each field in further detail. The input and output properties are the most crucial components as they constitute the base of the comparison. Precisely, the input contains the concrete value(s) used to reach the corresponding output value(s) of the functionality. This pair allows the script to identify the functionality. The output type is useful to compare the output obtained with angr and the expected value. Indeed, to resolve the concrete value from the result state memory, we need to know the exact type that we are looking for. The next field is the prototype, it is useful to generate a call state and callable matching the searched functionality. The next two attributes, specific types, and specific types values, are linked to each other. Actually, there is the same number of boolean values in the specific types as in the input as each boolean says if the input at the same index in the list is a custom type, such as a structure. When one value is set to true, then the specific types values list holds the value of this structure as a dictionary. Finally, the user hook field holds the name of the custom hook module in the SEMA toolchain SCDG repository. The script uses this component when generating the hooks file used by the toolchain. It is a crucial component of the preprocessing step as it alleviates the workload on the analyst.

Field name	Short Explanation
Input	List of input(s) value(s)
Output	Expected output
Output type	Expected output type
Prototype	Prototype of the functionality
Specific types	List of boolean values to specify if an input value has a specific type (i.e. struct)
Specific types values	List of specific values if specific types is true
User hook	Name of the custom SimProcedure in SEMA plugin hooks if it exists

Table 3.1: Table with the different fields of each functionality in the database.

We designed a methodology to populate our database with its first elements. Figure 3.2 illustrates the whole process. The first step was to identify relevant deterministic functionalities. In the context of this work, we used the *generate_domain* from the RAMDO malware similarly to the authors of SimID [2]. We also added CRC32, MurmurHash, a *copy_mem* function, and multiple utility functions from the GonnaCry malware. We chose those functions for various reasons. Regarding CRC32, MurmurHash, and the *copy_mem* function, the SEMA toolchain already tries to hook those functions but it relies on an inefficient pattern-matching approach. The latest results from the fact that slight modifications in implementation result in the need for a new pattern. Therefore, we can compare our approach with what is already integrated. As to the multiple utils functions from GonnaCry, we decided to select them to display another use case with malware. We will show that different compilations of the malware do not hinder the detection capacity of our tool.

The second step was to rewrite the algorithm, or at least find an implementation, of the corresponding algorithm. This step is crucial for our next step which consists in the generation of the actual input-output pair.

For the next step, we choose an input and run the algorithm with it to get its output. The deterministic relation between the input and output ensures that we will be able to retrieve the same pair if a binary implements the same functionality.

Finally, we add the functionality with the chosen input and the resulting output to the database. Rewriting the algorithm allows the analyst to understand its intricacies and renders the writing of a hook easier.

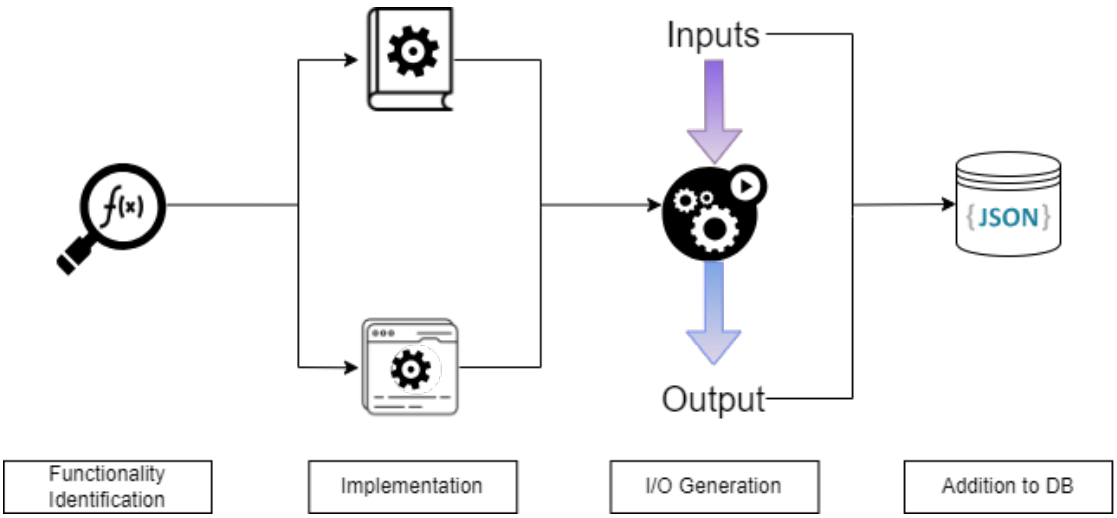


Figure 3.2: Our methodology to generate I/O values of functionalities

Even though this methodology has to be done manually and can be time-consuming, the gains of using hooks within the first step of the SEMA toolchain are worth the hassle of generating those signatures as shown in [4].

3.3 Preprocessing Step of the SEMA Toolchain

The main advantage of using our approach as a preprocessing step is that it reduces the workload of the symbolic execution. This results in a more efficient analysis for the SCDG part of the toolchain as it can explore more paths within the same time frame as shown in [4]. Moreover, it can also detect core functionalities of malware such as DGAs, hashing algorithms (as explained in section 2.4), or checksum routines, despite those being common obfuscated functions to avoid detection by other methods.

The results show that obfuscation techniques often used to hinder symbolic execution are inefficient against our tool. Therefore, it makes it complementary to the current toolchain to counter those techniques that are more and more common.

The integration with SEMA was pretty straightforward as it only depended on a new external file created by our version of SimID. We had to modify the hook plugin of the SEMA toolchain SCDG component to allow for custom hooking based on the results of the preprocessing step. Initially, the custom hooking mechanism could be turned on during the launch of the component. Then, it would try to hook every function for which a hook had been defined even if the functionality was not part of the binary. We added an optional parameter to the *hook* function so that one can give it a dictionary containing the project addresses to hook, the content of

each hook as a hexadecimal representation, the name of each hook, and each length. The latest indicates how many bytes of the original binary are covered by each hook. Moreover, it is important to note that the name of the hook should match the name of the corresponding standalone Python script in which the hook code is written. If the analyst gives the dictionary to the function, the plugin will take care of hooking the relevant addresses with the corresponding code. The dictionary is the result of the preprocessing step combined with the analyst work. It is saved inside a JSON file within the *scdg_ application* folder. Those modifications ensure that no changes are needed anywhere else in the toolchain code. Currently, only the first two modes can be used in combination with the SEMA toolchain as the third one would require a deeper modification of the internals of the SCDG analysis step.

Chapter 4

Results

In this section, we will start by showing that our approach is resilient to obfuscation techniques using our example toy from Section 2.3, and another dummy malware. We will show that different implementations of the same functionality are identified as the same feature in Section 4.1. Then, we will demonstrate that it can perform detection on real-world malware with the Warzone Remote Access Trojan in section 4.2. Finally, we will demonstrate the resilience of our approach to obfuscated real-world malware (GonnaCry) in Section 4.3. We decided to focus on the performance of our approach and not the advantages of using hooks in the SEMA toolchain as it was extensively explained in [4].

All tests are run on a Ubuntu 20.04 VM (kernel version 5.15.0-107) with 8 CPU cores and 8192 Mo RAM on a Windows 11 host (build number 10.0.22631) with a 12th Gen Intel(R) Core(TM) i7-12650H @ 2.30GHz CPU with 16 cores.

4.1 Countering Obfuscation

Binary	**Detected**	**Time** (hh:mm:ss:ms)	**Binary**	**Detected**	**Time** (hh:mm:ss:ms)
gcc	✓	00:00:01:567	ollvm-sub	✓	00:00:02:831
ollvm-fla	✓	00:00:02:831	tigress-vir	✓	00:00:05:849

Table 4.1: Results of the detection of the *transform* function for each version of our toy malware. Time is the time to complete analysis. *sub* = substitution, *fla* = flattening, *vir* = virtualization

In this section, we present the results of detection using the toy malware from

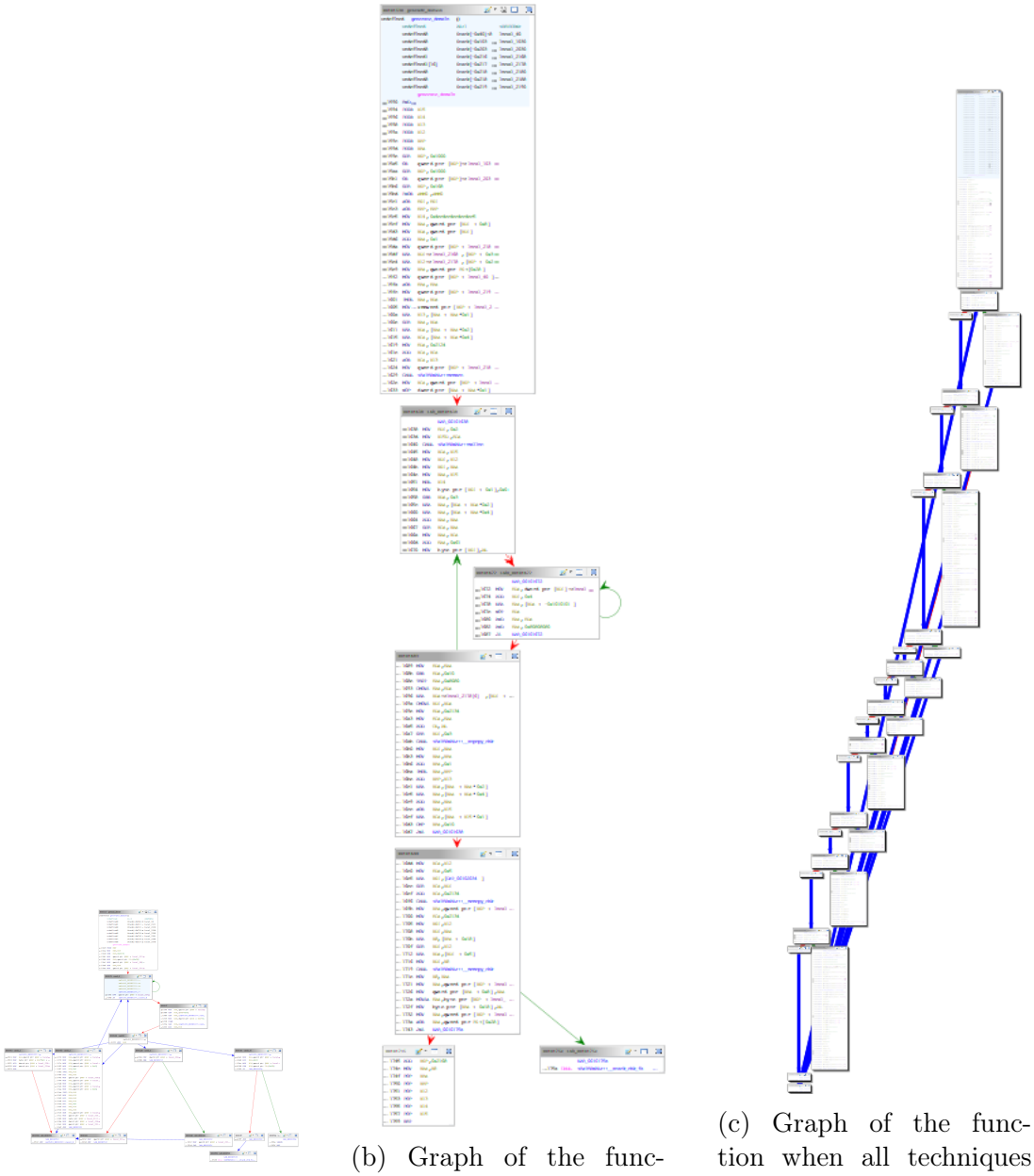
Binary	Detected	Time (hh:mm:ss)	Binary	Detected	Time (hh:mm:ss)
gcc-O0	✓	00:01:18	gcc-O1	×	00:02:02
gcc-O2	×	00:02:59	gcc-O3	×	00:03:22
musl-gcc-O0	✓	00:00:40	musl-gcc-O1	✓	00:00:45
musl-gcc-O2	✓	00:00:41	musl-gcc-O3	✓	00:00:58
ollvm-bcf	✓	00:00:52	ollvm-fla	✓	00:00:49
ollvm-sub	✓	00:00:48	ollvm-fla-bcf	✓	00:00:50
ollvm-sub-bcf	✓	00:00:48	ollvm-sub-fla	✓	00:00:50
ollvm-sub-fla-bcf	✓	00:00:52			
tigress-fla-O0	✓	00:06:27	tigress-fla-O1	✓	00:02:53
tigress-fla-O2	✓	00:03:03	tigress-fla-O3	✓	00:02:59
tigress-vir-O0	✓	00:37:37	tigress-vir-O1	✓	00:22:29
tigress-vir-O2	✓	00:20:10	tigress-vir-O3	✓	00:10:40
Average without virtualisation					00:01:47
Average					00:05:26

Table 4.2: Results of the detection of the domain generation algorithm of the Ramdo malware for each configuration and obfuscation. Time is the time to complete analysis. *bcf* = bogus control flow, *fla* = flattening, *sub* = substitution, *vir* = virtualization

earlier and the dummy malware from the original paper [2].

For our example toy, we will limit ourselves to the three obfuscated versions we used to show how pattern matching was beaten by obfuscation. The results are displayed in Table 4.1. Those results prove that our method works better than a pattern matching approach which would not detect the transform function in the obfuscated versions. We will now show that this technique works on a more complex example.

As can be seen in Table 4.2, our approach is able to detect functionalities in multiple binaries compiled with different obfuscation methods, compilation optimisation, and implementation choices. The detection is possible despite samples having radically diverging function graphs as shown in Figure 4.1 for the *gener-*



(a) Graph of the original unaltered function

(b) Graph of the function when binary compiled with GCC 9 at O3 level

(c) Graph of the function when all techniques of obfuscator-llvm are applied to the binary

Figure 4.1: Three graphs of the same functionality with diverging parameters (optimization and obfuscation)

ate_domain function of the Ramdo malware domain generation algorithm. More graphs for this functionality are available in Appendix D.2.

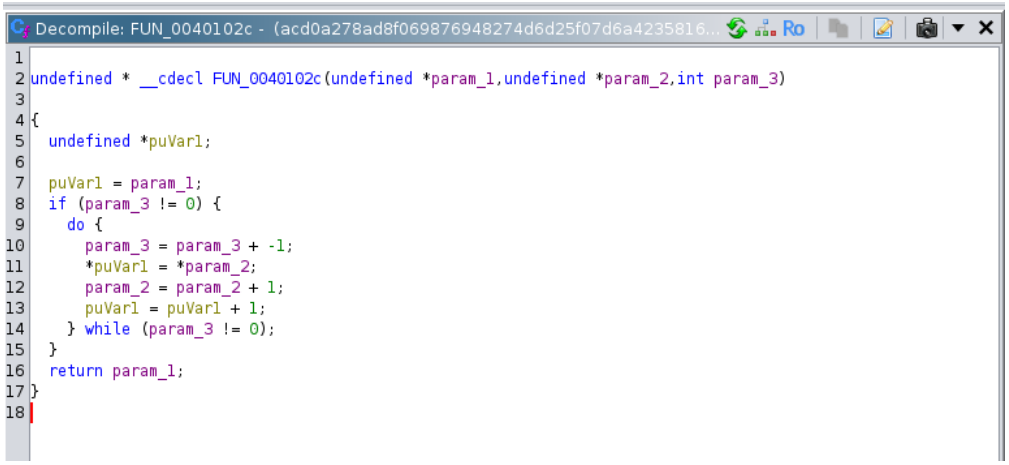


Figure 4.2: The memory copying function in Warzone sample.

This version of SimID did not detect the functionality when compiled with GCC at optimisation levels 1 to 3 but it did detect it when using musl-gcc. It might be due to the original GCC breaking something when trying to optimise whereas musl, which uses static linking, performs a more conservative approach and does not.

Interestingly, when the binary is compiled only with musl-gcc, our tool takes an increasing amount of time which can be related to the increase in the optimisation level. However, we can see that this does not hold when the binary is obfuscated with virtualisation where the increasing optimisation level is correlated to a lower analysis time. This relation might be due to the fact that higher optimisation mechanisms already counter some parts of the obfuscation inserted.

Our version of SimID detects the searched functionality in almost all cases including all obfuscated samples with an average detection time of 5 minutes 26 seconds. The average is brought up by the samples obfuscated with virtualisation which remains one of the most efficient counter-analysis methods. Depending on the analyst's needs, SimID could be fine-tuned to stop looking at a sample if it takes too long. However, since we currently want to use it at a preprocessing stage, it is still suitable despite taking a longer time. Moreover, virtualisation is a rare obfuscation technique. Therefore, our approach still fits the requirements to act as a preprocessing step of the SEMA toolchain as the average time without taking virtualised samples into account is only 1 minute and 47 seconds.

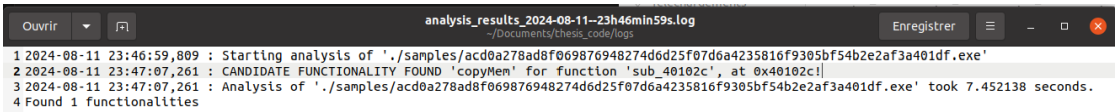


Figure 4.3: Our version of SimID identifies the *copy_mem* functionality.

4.2 First Use Case - Warzone RAT

For the Warzone RAT, we intended to prove that we could detect functions like CRC32, MurmurHash, or *copy_mem* as they all are excellent candidates for detection. However, due to a lack of computing resources, our version of SimID could not analyse all functions of those binaries and got stuck. The analysis explored about 10% of the total amount of functions without finding the functionalities. Despite this setback, we are confident that with more computing power, detection of those functionalities in Warzone binaries could become trivial. This affirmation is motivated by two main observations.

First, the analysis succeeds on the early functions of the binary before getting stuck. By succeeding, we mean that the analysis discards them because they do not match the function we are looking for. It is an encouraging sign that the analysis functions on this kind of malware with the hardware remaining the only challenge. If the problem was the type of malware, it would not be able to discard them in the first place.

The other observation, and the most important, is that we can confirm the presence of the *copy_mem* functionality using our third mode of SimID. When manually analysing the Warzone binary[1] using Ghidra, we identify *FUN_0040102c* as a function performing a memory copying operation. This is due to its specific architecture as can be seen in Figure 4.2. Then, using our last simID mode with this function as input, it is able to detect the *copy_mem* functionality expected as can be seen in Figure 4.3. This last fact proves that our approach works on real-world samples.

Therefore, we can assert that our strategy works on real-world malware, such as Warzone, but has been held back by the hardware available.

[1]sample with hash acd0a278ad8f069876948274d6d25f07d6a4235816f9305bf54b2e2af3a401df (available on VirusTotal)

4.3 Second Use Case - Resilience Against Obfuscation on GonnaCry Ransomware

The second use case is articulated around the GonnaCry ransomware. In this section, we will show that our tool is capable of detecting multiple functionalities even in the presence of obfuscation. Using the source code available here, we compiled different versions of the binary. Some of them are obfuscated with OLLVM. The primary goal of this experiment is to show that obfuscated malware samples can be analysed, and functionalities detected. However, there is another objective. This use case displays the efficiency of the second mode of the new version of SimID. With this mode, we look for a potential match for each functionality of the database. It results in a longer analysis time for each sample because it does not stop when one match is found. Table 4.3 displays the results of this analysis. First, the second column shows if a functionality was detected within the sample. Then, there is the number of functionalities detected if relevant. Finally, the total analysis time of the sample is in the last column.

The results are encouraging as only one functionality was not identified, *getFilenameExt*, even though present. All the others are detected in every sample no matter the obfuscation method used. This experiment shows that our approach can qualify for real-world usage. It is pretty efficient and is fast enough to run before a deeper analysis using SEMA. However, as seen in section 4.2, it requires a good amount of memory and computing power for some malware.

Binary	**Detected**	**# Functionalities**	**Time** (hh:mm:ss)
gcc-O0	✓	3	00:05:05
gcc-O1	✓	3	00:05:04
gcc-O2	✓	4*	00:06:04
gcc-O3	✓	4*	00:05:59
ollvm-bcf	✓	6*	00:03:12
ollvm-fla	✓	6*	00:03:44
ollvm-sub	✓	6*	00:06:28
ollvm-fla-bcf	✓	6*	00:03:40
ollvm-sub-bcf	✓	6*	00:07:02
ollvm-sub-fla	✓	6*	00:08:59
ollvm-sub-fla-bcf	✓	6*	00:13:22
Average number of functionalities detected			3**
Average analysis time			00:06:04

Table 4.3: Results of the full analysis of GonnaCry malware for binaries with various configurations and obfuscations. Time is the time to complete analysis. *bcf* = bogus control flow, *fla* = flattening, *sub* = substitution.
* means that at least one functionality among the ones detected is a duplicate.
** When removing duplicates

Chapter 5

Future Work

The main limitation of our approach is that it can be countered by splitting a function implementing a searched functionality into two, or more, parts. This means that the output of the first function is used as the input of the second function to reach the desired output. However, our method does not support the function splitting part which means that it won't detect the functionality despite it being present within the binary. An approach to solve this could be based on def-use pairs to track if the output of a first function directly becomes the input of another function.

Another improvement to have a more fine-tuned analysis could be to take side effects that do not result in any modification of the output into account. Currently, it sits as one of the main limitations because our approach is focused on the inner manipulation of input. Considering side effects could help create a more precise signature database. Moreover, it could provide valuable information to analysts when trying to analyse a binary by providing more information regarding its global actions.

In the case of implicit input written previously to memory, the authors of [2] claim that SimID cannot manage implicit input written to memory before the function. They add that tracking memory operations on uninitialised addresses could lead to further improvements.

Conditional execution, more specifically input check bypassing, which is an obfuscation method could pose a threat to our approach. Indeed, we rely on previously chosen input to detect a specific behaviour. Therefore, if it detects the same functionality, then the behaviour should be consistent with the same input. If the result differs then the functionality will not be detected. However, considering an attacker has access to our chosen input value, it could devise a specific path leading to another output preventing detection. This could be countered by

generating a random input/output pair following specific rules defined previously. Another idea could be to generate enough random inputs to reach a threshold coverage of the function.

A last amelioration for the SEMA toolchain would be to use SimID only on functions taking a lot of time during the analysis for the SCDG. The new version of SimID supports this approach but it needs to be implemented directly into the code of the toolchain which was out of scope for this work. Therefore, we leave this for future work.

Chapter 6

Conclusion

We started this thesis by reviewing various malware analysis techniques. Then, we thoroughly explored the problem of function clone detection, its implication for malware analysis, and numerous approaches to tackle it. Next, we extensively outlined the difficulties and the evolution of obfuscation and evasion techniques. At the same time, we advocated for a resilient approach based on the identification of input-output pairs, SimID.

We explained how we extended SimID by implementing new modes to fulfill different needs. We presented a novel signature database for function input and output pairs to combine with the newly adopted modularity. Moreover, we showcased a methodology to populate our database with various functionalities. The last axis of our contribution is the presentation of the distinct advantages of using SimID as a preprocessing step of the SEMA toolchain and the explanation of its integration.

Subsequent tests on fake and real malware showed the efficiency of the various modes to counter obfuscation techniques. It showcased that an approach like SimID is more reliable and less time-consuming than pattern-matching in addition to being resilient to all obfuscation techniques we tried.

Finally, we discussed the limitations of our work and expressed some opportunities for improvement.

In conclusion, we designed a successful function detection approach resilient to obfuscation to use as a preprocessing step of the SEMA toolchain which will in turn improve its performance.

Bibliography

[1] Di Freeze. *Cybercrime To Cost The World $9.5 trillion USD annually in 2024*. en-US. Section: Blogs. Oct. 2023. URL: https://cybersecurityventures.com/cybercrime-to-cost-the-world-9-trillion-annually-in-2024/ (visited on 01/19/2024).

[2] Sebastian Schrittwieser et al. "Obfuscation-Resilient Semantic Functionality Identification Through Program Simulation". en. In: *Secure IT Systems*. Ed. by Hans P. Reiser and Marcel Kyas. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 273–291. DOI: 10.1007/978-3-031-22295-5_15.

[3] Charles-Henry Bertrand Van Ouytsel and Axel Legay. "Malware Analysis with Symbolic Execution and Graph Kernel". en. In: *Secure IT Systems*. Ed. by Hans P. Reiser and Marcel Kyas. Cham: Springer International Publishing, 2022, pp. 292–310. DOI: 10.1007/978-3-031-22295-5_16.

[4] Serena Lucca, Christophe Crochet, and Axel Legay. "On Exploiting Symbolic Execution to Improve the Analysis of RAT Samples with angr". en. In: ().

[5] Anitta Patience Namanya et al. "The World of Malware: An Overview". In: *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. Aug. 2018, pp. 420–427. DOI: 10.1109/FiCloud.2018.00067.

[6] Manuel Egele et al. "A survey on automated dynamic malware-analysis techniques and tools". In: *ACM Computing Surveys* 44.2 (Mar. 2008), 6:1–6:42. DOI: 10.1145/2089125.2089126.

[7] Fabrizio Biondi et al. "Tutorial: An Overview of Malware Detection and Evasion Techniques". en. In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 565–586. DOI: 10.1007/978-3-030-03418-4_34.

[8] Luca Caviglione et al. “Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection”. In: *IEEE Access* 9 (2021). Conference Name: IEEE Access, pp. 5371–5396. DOI: 10.1109/ACCESS.2020.3048319.

[9] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Computing Surveys* 51.3 (May 2018), 50:1–50:39. DOI: 10.1145/3182657.

[10] Irfan Ul Haq and Juan Caballero. “A Survey of Binary Code Similarity”. In: *ACM Computing Surveys* 54.3 (Apr. 2021), 51:1–51:38. DOI: 10.1145/3446371.

[11] Andrew Walenstein and Arun Lakhotia. “The Software Similarity Problem in Malware Analysis”. en. In: *DROPS-IDN/v2/document/10.4230/DagSemProc.06301.14* (2007). Publisher: Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/DagSemProc.06301.14.

[12] Brenda S Baker, Udi Manber, and Robert Muth. “Compressing Differences of Executable Code”. en. In: *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*. 1999, pp. 1–10.

[13] Zheng Wang, Ken Pierce, and Scott McFarling. “BMAT – A Binary Matching Tool for Stale Profile Propagation”. en. In: *Journal of Instruction-Level Parallelism* 2 (2000), pp. 1–20.

[14] Christopher Kruegel et al. “Polymorphic Worm Detection Using Structural Information of Executables”. en. In: *Recent Advances in Intrusion Detection*. Ed. by Alfonso Valdes and Diego Zamboni. Berlin, Heidelberg: Springer, 2006, pp. 207–226. DOI: 10.1007/11663812_11.

[15] Debin Gao, Michael K. Reiter, and Dawn Song. “BinHunt: Automatically Finding Semantic Differences in Binary Programs”. en. In: *Information and Communications Security*. Ed. by Liqun Chen, Mark D. Ryan, and Guilin Wang. Berlin, Heidelberg: Springer, 2008, pp. 238–255. DOI: 10.1007/978-3-540-88625-9_16.

[16] Andrea Marcelli et al. “How Machine Learning Is Solving the Binary Function Similarity Problem”. en. In: 2022, pp. 2099–2116.

[17] Steven H. H. Ding, Benjamin C. M. 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)*. ISSN: 2375-1207. May 2019, pp. 472–489. DOI: 10.1109/SP.2019.00003.

[18] Peihua Zhang et al. “Khaos: The Impact of Inter-procedural Code Obfuscation on Binary Diffing Techniques”. In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2023. New York, NY, USA: Association for Computing Machinery, Feb. 2023, pp. 55–67. DOI: 10.1145/3579990.3580007.

[19] Xuezixiang Li, Yu Qu, and Heng Yin. “PalmTree: Learning an Assembly Language Model for Instruction Embedding”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 3236–3251. DOI: 10.1145/3460120.3484587.

[20] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. en. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies*. Vol. 1. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[21] Luca Massarelli et al. “Function Representations for Binary Similarity”. In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (July 2022). Conference Name: IEEE Transactions on Dependable and Secure Computing, pp. 2259–2273. DOI: 10.1109/TDSC.2021.3051852.

[22] Michael Pucher, Christian Kudera, and Georg Merzdovnik. “Detecting Obfuscated Function Clones in Binaries using Machine Learning”. en. In: *Proceedings 2022 Workshop on Binary Analysis Research*. San Diego, CA, USA: Internet Society, 2022. DOI: 10.14722/bar.2022.23005.

[23] 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)*. ISSN: 2375-1207. May 2023, pp. 2375–2390. DOI: 10.1109/SP46215.2023.10179439.

[24] Kenneth Brezinski and Ken Ferens. “Metamorphic Malware and Obfuscation: A Survey of Techniques, Variants, and Generation Kits”. en. In: *Security and Communication Networks* 2023 (Sept. 2023). Ed. by Konstantinos Rantos, pp. 1–41. DOI: 10.1155/2023/8227751.

[25] *Cascade virus*. en-US. URL: https://encyclopedia.kaspersky.com/knowledge/cascade-virus/ (visited on 04/15/2024).

[26] Ilsun You and Kangbin Yim. “Malware Obfuscation Techniques: A Brief Survey”. In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. Nov. 2010, pp. 297–300. DOI: 10.1109/BWCCA.2010.85.

[27] *Evasion techniques*. URL: https://evasions.checkpoint.com/about/ (visited on 02/04/2024).

[28] Pascal Junod et al. "Obfuscator-LLVM – Software Protection for the Masses". In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. May 2015, pp. 3–9. DOI: 10.1109/SPRO.2015.10.

[29] Christian Collberg et al. "Distributed application tamper detection via continuous software updates". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. New York, NY, USA: Association for Computing Machinery, Dec. 2012, pp. 319–328. DOI: 10.1145/2420950.2420997.

[30] *Features · obfuscator-llvm/obfuscator Wiki*. URL: https://github.com/obfuscator-llvm/obfuscator/wiki/Features (visited on 01/25/2024).

[31] *Tigress Obfuscator Transformations*. URL: https://tigress.wtf/transformations.html (visited on 01/25/2024).

[32] *Tigress Obfuscator*. URL: https://tigress.wtf/encodeArithmetic.html (visited on 02/03/2024).

[33] Lingxiao Jiang and Zhendong Su. "Automatic mining of functionally equivalent code fragments via random testing". In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ISSTA '09. New York, NY, USA: Association for Computing Machinery, July 2009, pp. 81–92. DOI: 10.1145/1572272.1572283.

[34] 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*. CCS '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 169–182. DOI: 10.1145/2382196.2382217.

[35] Yikun Hu et al. "Binary Code Clone Detection across Architectures and Compiling Configurations". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. May 2017, pp. 88–98. DOI: 10.1109/ICPC.2017.22.

[36] Jannik Pewny et al. "Cross-Architecture Bug Search in Binary Executables". In: *2015 IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2015, pp. 709–724. DOI: 10.1109/SP.2015.49.

[37] *angr*. URL: https://angr.io/ (visited on 12/20/2024).

[38] *Intermediate Representation - angr documentation*. URL: https://docs.angr.io/en/latest/advanced-topics/ir.html (visited on 01/04/2024).

[39] *valgrind-vex/pub/libvex_ir.h at master smparkes/valgrind-vex*. en. URL: https://github.com/smparkes/valgrind-vex/blob/master/pub/libvex_ir.h (visited on 02/01/2024).

[40] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[41] Charles-Henry Bertrand Van Ouytsel et al. "Tool Paper - SEMA: Symbolic Execution Toolchain for Malware Analysis". en. In: *Risks and Security of Internet and Systems*. Ed. by Slim Kallel et al. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 62–68. DOI: 10.1007/978-3-031-31108-6_5.

[42] *csvl/SEMA-ToolChain*. original-date: 2022-03-11T08:36:34Z. May 2024. URL: https://github.com/csvl/SEMA-ToolChain.

[43] Yan Shoshitaishvili et al. "angr: A Powerful and User-friendly Binary Analysis Platform". en. In: (2017).

[44] *angr internals*. URL: https://angr.io/blog/throwing_a_tantrum_part_1/ (visited on 05/31/2024).

[45] Daniel Plohmann et al. "A Comprehensive Measurement Study of Domain Generating Malware". en. In: 2016, pp. 263–278.

[46] S. Koppier. "The Path Explosion Problem in Symbolic Execution: An Approach to the Effects of Concurrency and Aliasing". en. Accepted: 2020-05-26T18:00:11Z. MA thesis. 2020.

[47] *What is Remote Access Trojan (RAT)?* en-US. URL: https://www.checkpoint.com/cyber-hub/threat-prevention/what-is-remote-access-trojan/ (visited on 07/10/2024).

[48] *What Is Warzone RAT?* en. URL: https://www.blackberry.com/us/en/solutions/endpoint-security/ransomware-protection/warzone (visited on 07/10/2024).

[49] Roberto Baldoni et al. "Assisting Malware Analysis with Symbolic Execution: A Case Study". en. In: *Cyber Security Cryptography and Machine Learning*. Ed. by Shlomi Dolev and Sachin Lodha. Cham: Springer International Publishing, 2017, pp. 171–188. DOI: 10.1007/978-3-319-60080-2_12.

[50] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". en. In: *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. DOI: 10.14722/ndss.2016.23368.

[51] Yan Shoshitaishvili et al. “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware”. en. In: *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015. DOI: 10.14722/ndss.2015.23294.

[52] Tarcísio Marinho. *How ransomware work's and GonnaCry linux ransomware*. en. Aug. 2018. URL: https://medium.com/@tarcisioma/how-ransomware-works-and-gonnacry-linux-ransomware-17f77a549114 (visited on 07/10/2024).

Appendix A

SimID algorithms

Algorithm A.1: Actual implementation of SimID in pseudocode

Input: P (Program in binary representation)
Input: EIN (Expected INput)
Input: EOUT (Expected OUTput)
Input: EPROTO (Expected function PROTOtype)
Output: CF (list of Candidate Functions implementing the searched functionality

```
 1 CFG ← reconstructControlFlowGraph(P)
 2 for function in P as FUNC do
 3 |   CC ← reconstructCallingConvention(FUNC, CFG)
 4 |   if CC.PROTOTYPE = EPROTO ∨ (analyze_if_unkwnow_prototype
   |    ∧ CC.PROTOTYPE = None) then
 5 |   |   CS ← generateCallState(FUNC, EIN, EPROTO)
 6 |   |   C ← createCallable(FUNC, EPROTO, CS)
 7 |   |   RET ← C(EIN)
 8 |   |   if RET = EOUT then
 9 |   |   |   CF ← CF + FUNC
10 |   else
   |   |   // Go directly to next iteration
11 |   |   continue
12 return CF
```

Algorithm A.2: General concept of SimID in pseudocode from [2]

Input: P (Program in binary representation)
Input: EIN (Expected INput)
Input: EOUT (Expected OUTput)
Input: EPROTO (Expected function PROTOtype)
Output: CF (list of Candidate Functions implementing the searched functionality

```
1  CFG ← reconstructControlFlowGraph(P)
2  for function in P as FUNC do
3  |  CC ← reconstructCallingConvention(FUNC, CFG)
4  |  if CC.PROTOTYPE ∨ CC.PROTOTYPE = EPROTO then
5  |  |  CS ← generateCallState(FUNC, EIN, EPROTO)
6  |  |  C ← createCallable(FUNC, EPROTO, CS)
7  |  |  RET ← C(EIN)
8  |  |  if RET = EOUT then
9  |  |  |  CF ← CF + FUNC
10 |  else
   |  |  // List of "wrong" function prototypes
11 |  |  WFP ← WFP + FUNC
12 if CF is empty then
13 |  for function in WFP as func do
14 |  |  CS ← generateCallState(FUNC, EIN, EPROTO)
15 |  |  C ← createCallable(FUNC, EPROTO, CS)
16 |  |  RET ← C(EIN)
17 |  |  if RET = EOUT then
18 |  |  |  CF ← CF + FUNC
19 return CF
```

Appendix B

History of Virus and Obfuscation Techniques

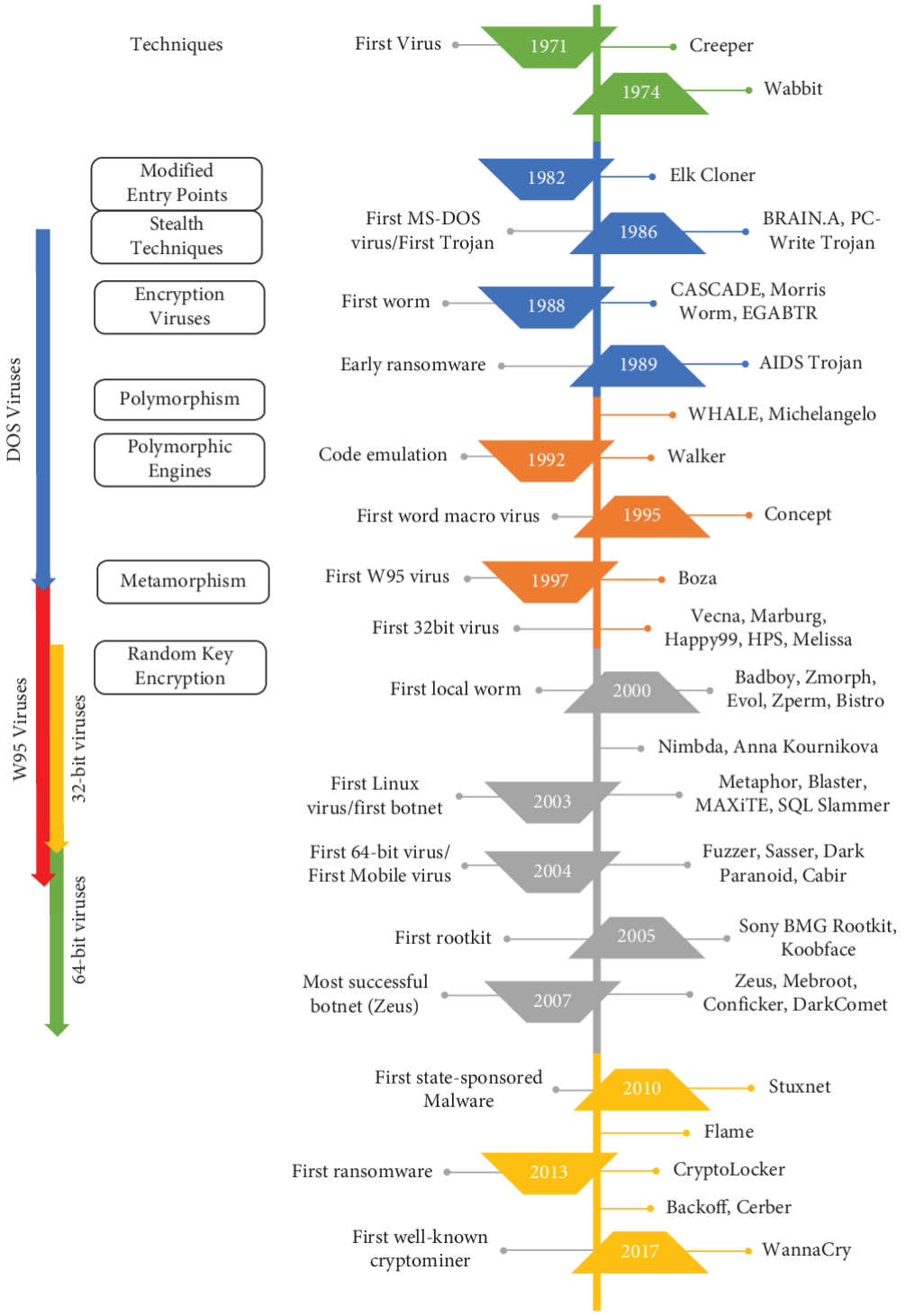


Figure B.1: Timeline of virus and obfuscation techniques from [24]

Appendix C

Obfuscator-LLVM Bogus Control Flow

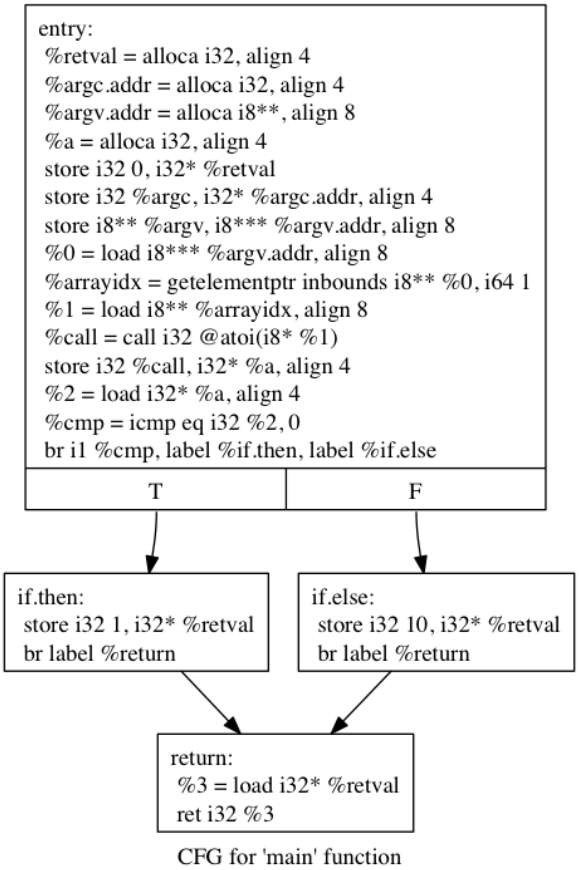


Figure C.1: Initial version of the CFG before obfuscation [30]

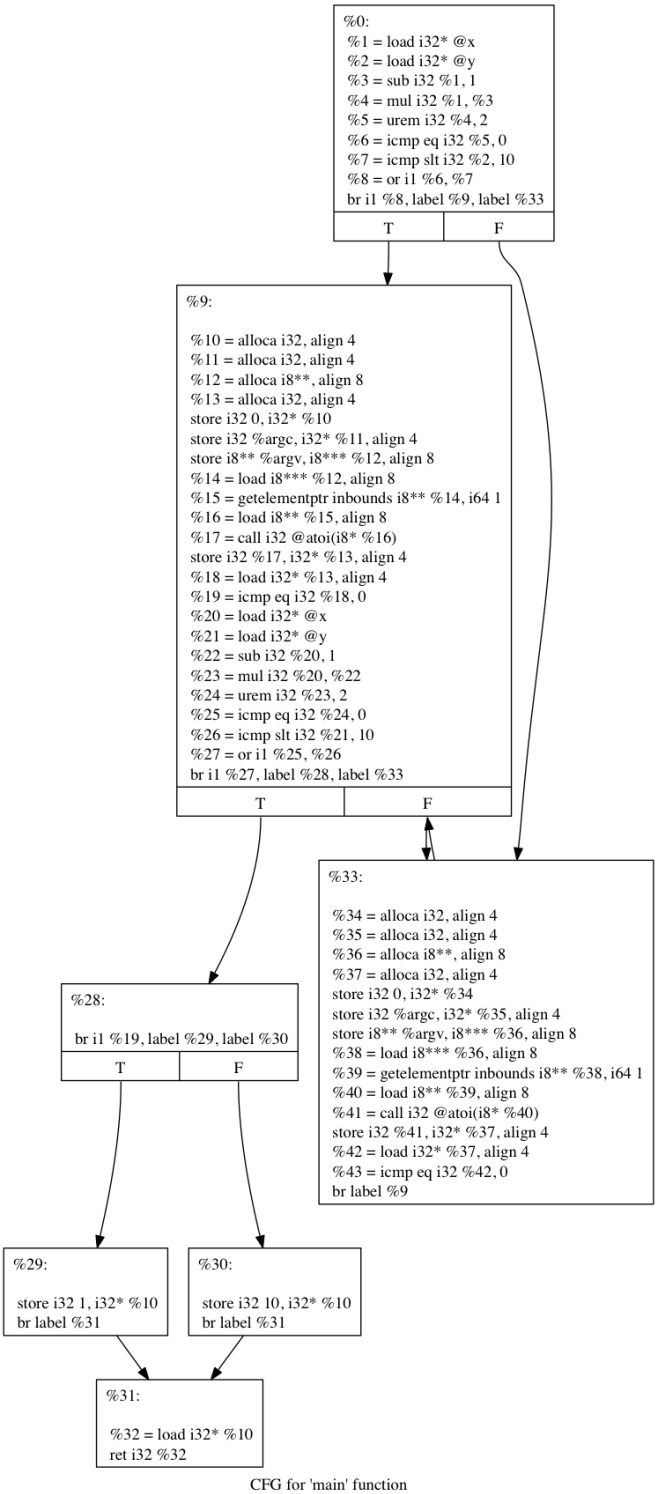
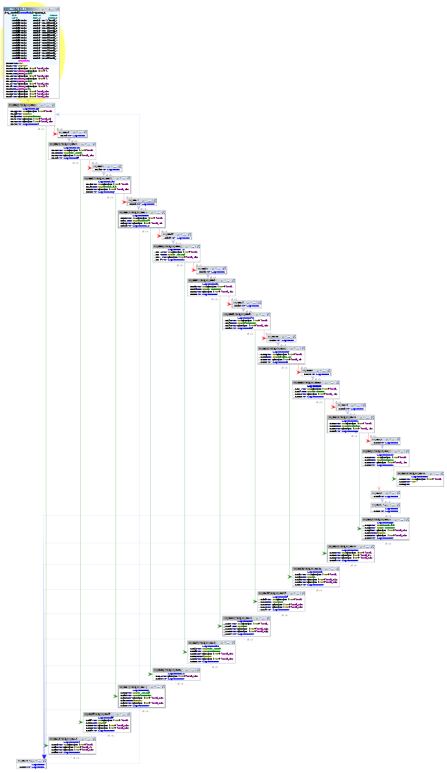


Figure C.2: CFG after *Bogus Control Flow* obfuscation [30]

Appendix D

Functions Graphs

D.1 Toy example - transform



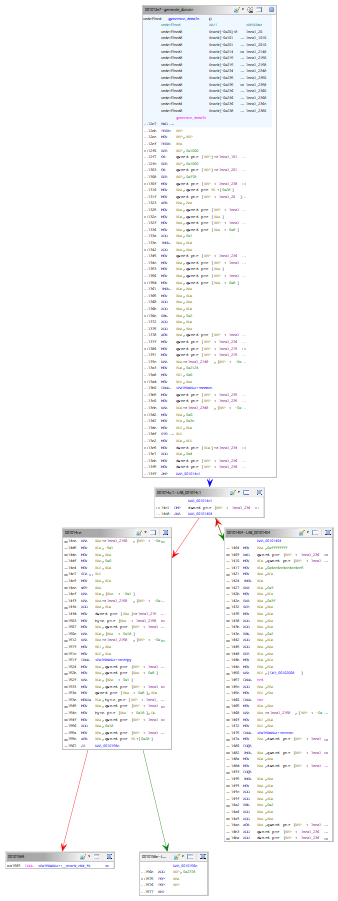
(a) Graph of the *transform* function when flattening from obfuscator-llvm is applied to the binary

```
00401130 - transform
int __stdcall transform(uint param_1)
      int               EAX:4          <RETURN>
      uint              EDI:4          param_1
      undefined4        Stack[-0xc]:4  local_c
      undefined4        Stack[-0x10]:4 local_10
      undefined4        Stack[-0x14]:4 local_14
      undefined4        Stack[-0x18]:4 local_18
      undefined4        Stack[-0x1c]:4 local_1c
      undefined4        Stack[-0x20]:4 local_20
      undefined4        Stack[-0x24]:4 local_24
      undefined4        Stack[-0x28]:4 local_28
      undefined4        Stack[-0x2c]:4 local_2c
      undefined4        Stack[-0x30]:4 local_30
      undefined4        Stack[-0x34]:4 local_34
      undefined4        Stack[-0x38]:4 local_38
      undefined4        Stack[-0x3c]:4 local_3c
      undefined4        Stack[-0x40]:4 local_40
      undefined4        Stack[-0x44]:4 local_44
      undefined4        Stack[-0x48]:4 local_48
           transform
...1130 PUSH  RBP
...1131 MOV   RBP,RSP
...1134 MOV   dword ptr [RBP + local_10]...
...1137 MOV   param_1,dword ptr [RBP + l...
...113a AND   param_1,0xff
...1140 MOV   dword ptr [RBP + local_10]...
...1143 MOV   param_1,dword ptr [RBP + l...
...1146 XOR   param_1,0x9c
...114c MOV   dword ptr [RBP + local_10]...
...114f MOV   param_1,dword ptr [RBP + l...
...1152 SHL   param_1,0x1
...1155 MOV   dword ptr [RBP + local_10]...
...1158 MOV   dword ptr [RBP + local_14]...
...115f MOV   dword ptr [RBP + local_18]...
```

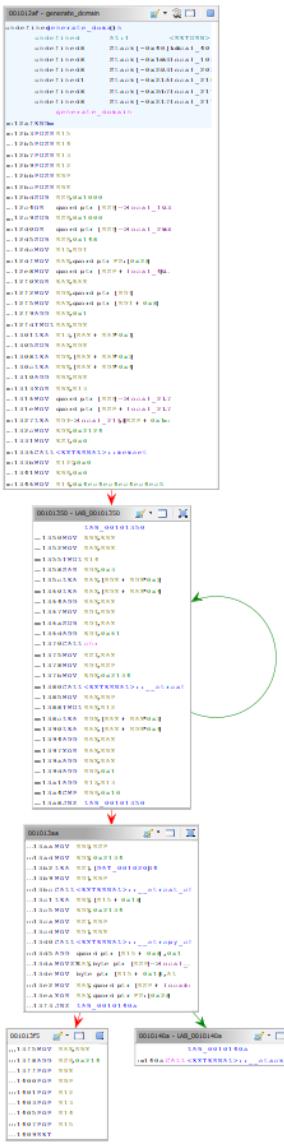
(b) Zoom on the first instructions of the function which are already different from the original version.

D.2 DGA RAMDO

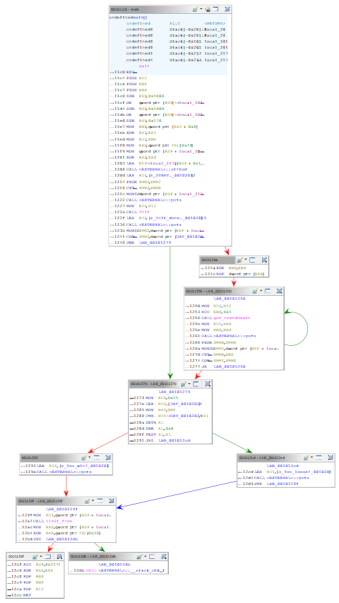
Following are the graphs for the *generate_ domain* function from the RAMDO malware.



(a) Binary compiled with gcc 9 at O0 level

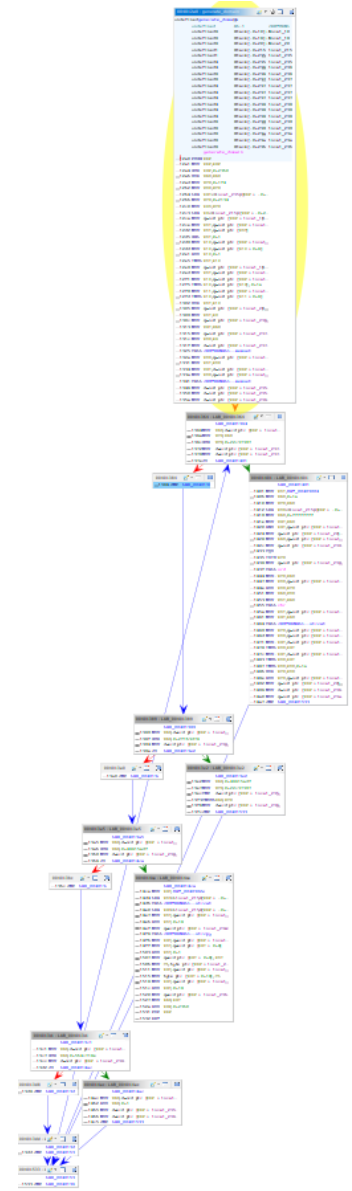


(b) Binary compiled with gcc 9 at O1 level

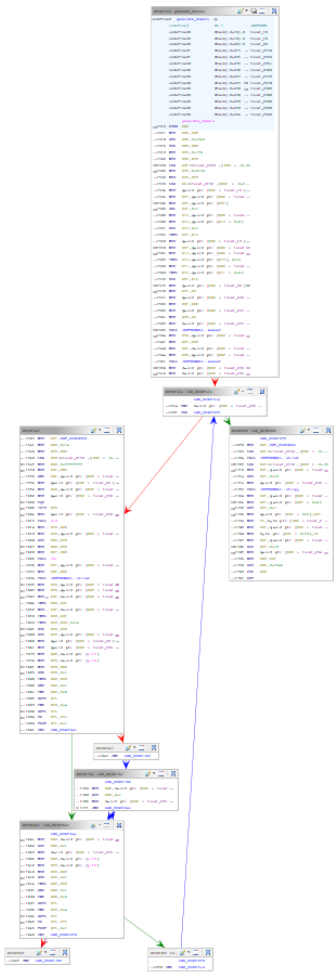


(c) Binary compiled with gcc 9 at O2 level

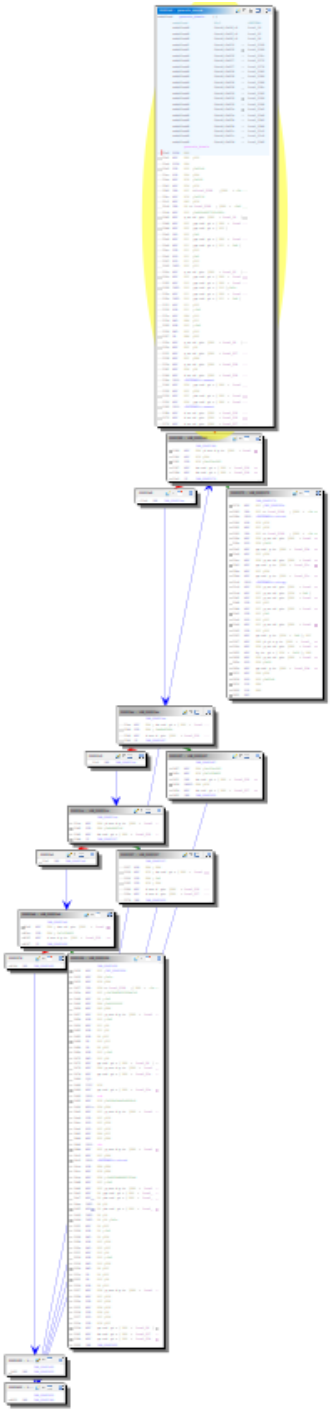
(a) Binary obfuscated with obfuscator-llvm substitution method



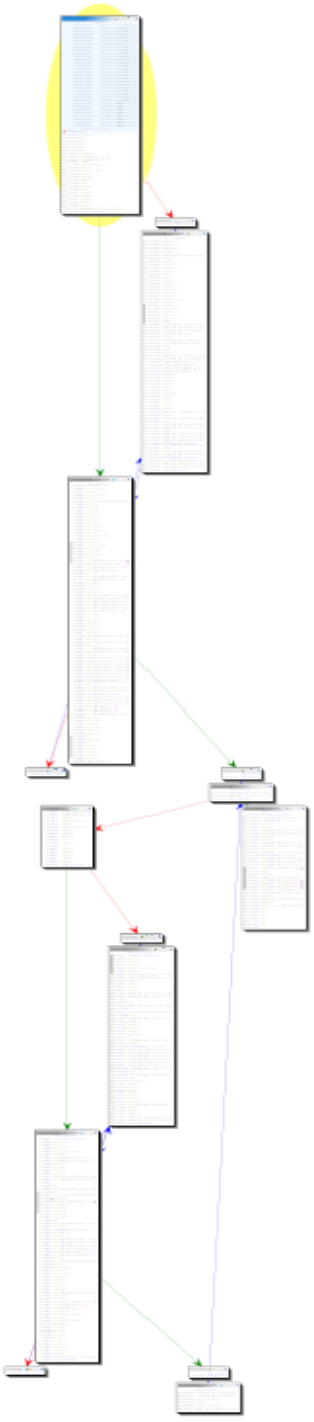
(b) Binary obfuscated with obfuscator-llvm flattening method



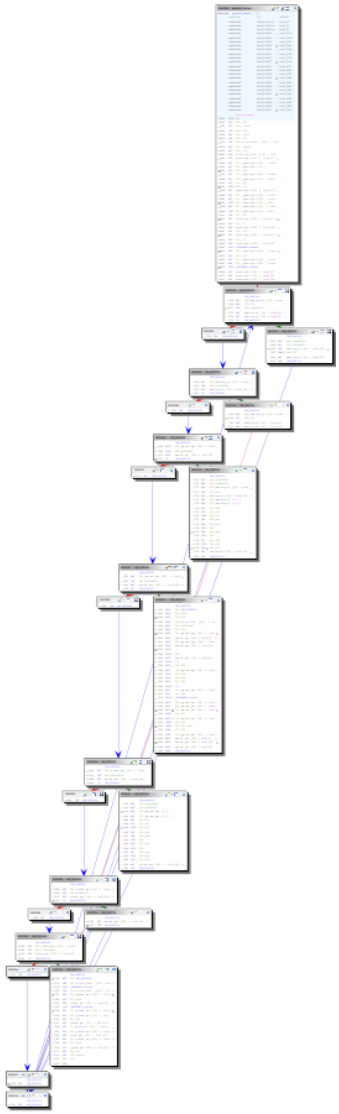
(c) Binary obfuscated with obfuscator-llvm bogus control flow method



(a) Binary obfuscated with obfuscator-llvm substitution & flattening methods



(b) Binary obfuscated with obfuscator-llvm substitution & bogus control flow methods



(c) Binary obfuscated with obfuscator-llvm flattening & bogus control flow methods

Appendix E

Code Repositories

The code for our version of SimID, the current database of functionalities, some test binaries, and logs can be found at : https://github.com/antoinemouchet/SimID-extended

The code for the integration of SimID as a preprocessing step of SEMA can be found at : https://github.com/antoinemouchet/SEMA/tree/preprocessing-simid

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