UCLouvain

epl

École polytechnique de Louvain

Enhancing Malware Analysis with Concolic Execution

Author: **Thomas Bouvencourt**
Supervisor: **Kim Mens**
Readers: **Charles-Henry Bertrand Van Ouytsel, Charles Pecheur**
Academic year 2024–2025
Master [120] in Computer Science

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Kim Mens, and my reader, Professor Charles Pecheur, for their dedicated involvement in the supervision of this thesis, especially given the circumstances. In addition, I would like to thank the entire team of assistants, as well as Charles-Henry Bertrand Van Ouytsel, for their time and valuable advice. Finally, I would like to thank my friends and family for their support and patience during the long hours I spent trying to explain the purpose of my work.

Abstract

Malware is constantly evolving, making it increasingly difficult to analyze using traditional techniques. To keep up with these changes, new tools and methodologies are required. In this context, the Symbolic Execution for Malware Analysis (SEMA) toolchain was developed. By using symbolic execution, SEMA analyzes binaries to generate System Call Dependency Graphs (SCDGs), which can then be used to classify malware and assist in future detection. Symbolic execution offers many advantages but remains vulnerable to complex obfuscation techniques, including packing. In such cases, concolic execution (a hybrid approach combining concrete and symbolic execution) allows the unpacking phase to be executed concretely before turning into symbolic state.

This master's thesis enhances SEMA by integrating concolic execution using Symbion, a component of the angr framework. Despite its limitations, this study establishes a foundation for future enhancements and research involving concolic execution within SEMA.

Use of AI

The use of AI in this work complies with the rules established by the Université catholique de Louvain, and more specifically by the École Polytechnique de Louvain (EPL). In agreement with my supervisor, AI was used as a support tool for writing assistance, such as spelling correction, rephrasing suggestions, but was never used to generate any part of the content of this thesis. It was also used to assist in the understanding of certain bugs and to provide comments on specific portions of code.

Contents

Chapter 1

Introduction

The cybercrime group known as Golden Chickens (also tracked as TA4557 or Venom Spider) continues to expand its malware arsenal with the development of two new tools: TerraStealerV2 and TerraLogger. These latest threats, revealed in early 2025 [19], demonstrate the group's persistent innovation in credential theft and stealth tactics.

This is just one of many threat updates that appear in the world of cybersecurity. The constant stream of new malware, updated tools, and evolving attack methods like zero days, highlights how dynamic this field is. In such an environment, research and vigilance are not an option, they are essential. Studying malware isn't just about understanding existing threats, it's about keeping environment, safe with a volatile ecosystem that challenges defenders at every step.

1.1 Context

Malicious software (malware) is detected every day and is becoming increasingly widespread. The emergence of new variants and the use of advanced evasion techniques are challenging current approaches. Traditional detection methods, mainly based on static analysis and signature matching, are often outdated and no longer sufficient. It is therefore essential to develop new methods capable of detecting and classifying malware and its variants, in order to defend ourselves as efficiently as possible.

Several techniques already exist for analyzing different types of malware. As explained above, static analysis (Section 2.2.1) is one of the oldest approaches and remains an effective first step. In addition, many dynamic techniques have emerged to more effectively detect and classify malware, especially those that employ evasion strategies or rewrite their code during execution. To address the proliferation of malware, tools like `angr` [30] have emerged. By using symbolic

analysis (Section 2.2.3), These tools can analyze malware dynamically and achieve greater coverage. Our focus will mainly be on symbolic execution, and more specifically on concolic execution (a hybrid technique that merges the benefits of both concrete and symbolic execution) (Section 2.2.4). To achieve this goal, we will use SEMA (Section 2.4), a powerful tool designed for analyzing various malware families.

Finally, malware analysis is a challenging field that evolves constantly. Numerous techniques are used to hide the different behaviors of a program. It is therefore essential to continue studying this field in order to stay up to date and respond as quickly and effectively as possible to emerging threats.

1.2 Problem

In current research, one of the main challenges of symbolic execution is the path explosion problem. This occurs when the execution is represented as a tree whose size grows exponentially, making the analysis quickly too heavy in terms of resources. Several techniques aim to reduce the number of explored paths [18]. However, they often do so at the cost of losing important information. Another drawback is that malware is often packed in order to hide its signature, making analysis significantly more difficult [13].

Another major limitation of symbolic execution in achieving theoretical maximal coverage is its high time and resource consumption, particularly due to the many conditional structures and loops commonly found in complex programs. This makes exhaustive analysis difficult in realistic scenarios [4].

Moreover, modern malware is becoming increasingly complex, often using advanced techniques such as obfuscation, encryption, conditional logic, or packing. These strategies make both detection and analysis significantly more difficult.

1.3 Motivation

This work is driven by the goal of enhancing the capabilities of the SEMA (Symbolic Execution for Malware Analysis)[1] tool and exploring the strengths and limitations of concolic execution using Symbion [15], an extension of angr framework. Although concolic technique first emerged in the early 2000s, its optimal use remains an active area of research. Today, we see a growing adoption of this approach, often used in combination with other tools such as fuzzers [22] to achieve better performance in specific phases, especially in phases involving nested constraints.

[1] https://github.com/csvl/SEMA

Beyond improving SEMA itself, this research aims to highlight the respective advantages of symbolic and concrete execution. By combining these two approaches, it becomes possible to overcome some of their limitations. Additionally, this work establishes a foundation for future enhancements and research on concolic execution in SEMA.

Finally and as a summary, this project aims to achieve several specific goals, each contributing to a better understanding and implementation of concolic execution in the context of malware analysis:

- **Integrate concolic execution into SEMA:** This represents a core enhancement of the tool, enabling it to support hybrid execution analyses by combining symbolic and concrete execution. Also to create a foundation that can be reused and extended in future research, including the exploration of alternative concolic execution strategies or new analysis modules.
- **Packing analysis:** Provides a focused analysis of UPX[2] packed binaries to identify key addresses used for Symbion.

1.4 Approach

All the code discussed in this work can be found on GitHub: `https://github.com/tbouvencourt/TFE_Enhancing_Malware_Analysis_with_Concolic_Execution/tree/production`. To achieve these objectives, the following step by step approach will be followed:

- **Identify and reproduce an example from the documentation (Section 3.1):** Once an appropriate tool is identified, we reproduced an example provided in the documentation. This served as a foundation for implementing concolic execution within the tool.
- **Validation (Section 3.2):** Before integration, experiments on different sample will help validate the approach and confirm the practical applicability of the selected tools.
- **Integrate the concolic solution into SEMA (Section 3.3):** This step consists of merging the concolic execution mechanism within the existing architecture of SEMA.
- **Understanding UPX and identifying relevant addresses:** Using either static or dynamic analysis, attempt to examine the behavior of files packed with UPX.

[2]https://upx.github.io/

Chapter 2

Background Material

In this chapter, we introduce the theoretical background necessary for understanding the implementation of concolic execution in SEMA, which will be presented in Section 3. Section 2.1 introduces the context of Linux malware, including an overview of operating systems and malware families. Section 2.2 presents the two main approaches to malware analysis: static and dynamic analysis. Section 2.3 focuses on the angr framework, which is used as the symbolic execution engine in SEMA. We describe key concepts such as execution paths, symbolic variables, constraint solving, and the use of hooks and SimProcedures. Section 2.4 details the architecture and internal processes of the SEMA tool (Symbolic Execution-based Malware Analysis), including our proposed enhancements through concolic execution. Section 2.5 introduces Symbion[15], a tool designed to integrate concrete execution into symbolic workflows. Section 2.6 addresses the problem of packed binaries, which pose significant challenges to both static and dynamic analysis techniques. Finally, we examine additional tools that are relevant to hybrid execution (Section 2.7).

2.1 Linux malware

This section presents the environment and constraints under which the experiments in this work were conducted. The analysis and the symbolic execution were tested on Linux binaries, compiled for 64-bit architectures, and executed in a controlled Ubuntu 24.04.1 LTS environment. One main reason justifies the focus on Linux-based examples: the simplicity of the analysis. Working with binaries that are compiled and executed under Linux facilitates the use of debugging tools such as `gdb`, and overall makes it easier to understand and analyze binaries. Additionally, the tool, `Symbion` [15], used in our concolic execution approach, has not been

actively maintained since 2021[1]. As a result, several bugs occur when analyzing Windows binaries, particularly in interactions with concrete execution backends. These two factors led us to focus on Linux binaries for this study. Nevertheless, this choice remains a limitation of our work since the vast majority of malware binaries in the wild target the Windows platform.

2.1.1 Format

The binaries analyzed in this study follow the ELF (Executable and Linkable Format) [36], which is the default format for Unix/Linux systems. This format is particularly relevant in symbolic execution as its structure allows tools like `SEMA` to extract metadata for execution.

- **ELF Header**: Defines the file type, machine architecture, and the program's entry point.
- **Program Header Table**:Describes segments to be loaded into memory.
- **Section Header Table**: Describes each section of the binary, such as `.text`, `.data`, or `.bss`.

These elements are parsed during project creation (see `SemaSCDG.py`) and guide the construction of the internal model used by `angr`.

2.1.2 Families

A primary objective of malware analysis is to classify malicious samples and identify their corresponding families [26] . The classification process aims to determine the malware type by examining its behavior and general attributes to :

- Understand the malware's behavior.
- Develop effective detection signatures.
- Detect new variants.
- Enable appropriate countermeasures in case of an attack.

The detection of malware families remains an active area of research. Some malware or families of malware continue to be frequently detected in 2025[2], including RATs, ransomware, cryptocurrency miners, and info stealers. In addition to this

[1] `https://github.com/angr/angr/issues/2701`
[2] `https://www.cisecurity.org/insights/blog/top-10-malware-q1-2025`

work, other master's theses focus on the practical study of specific malware families, contributing to more effective analysis and detection through the SEMA framework [20]. In this work, no specific malware family is analyzed in detail due to the complexity of the main task, although we do focus on packed malware samples that may employ basic evasion techniques.

2.2 Malware Analysis

Although it is difficult, if not impossible, to fully determine the behavior of a malicious program, various techniques attempt to approach this goal by classifying behaviors and matching them with similar cases. To achieve this, several techniques exist for analyzing malware.

2.2.1 Static Analysis

Static analysis involves examining a program's content using techniques such as syntactic pattern matching [7]. These techniques allow examination of a program without executing it [14]. It allows the extraction of certain information, such as strings, function names, or general patterns. However, it does not directly test the program's behavior at runtime, as is the case with certain dynamic analysis techniques (see Section 2.2.2). Static analysis is particularly vulnerable to methods that conceal a program's true behavior, such as evasion techniques, packing, or the use of crypting techniques. Tools like Yara[3], Ghidra[4], and IDA Pro[5] allow analysts to inspect and reverse engineer binaries, detect known patterns or signatures, and extract valuable static information.

2.2.2 Dynamic Analysis

A distinguishing feature of dynamic analysis is that the malware is actually executed [7], this makes it resilient to various obfuscation techniques (string encryption). As previously mentioned, this introduces new and significant risks for the analyst or the software. To reduce the risks associated with executing malware, dynamic analysis is typically performed within virtual machines known as sandboxes (Cuckoo[6]). These sandboxes aim to emulate the behavior of a real operating system, including aspects such as file systems, network traffic, and memory activity. Moreover, modern malware is often capable of detecting when it is being analyzed and may

[3]https://virustotal.github.io/yara/
[4]https://github.com/NationalSecurityAgency/ghidra
[5]https://hex-rays.com/ida-pro
[6]https://github.com/cuckoosandbox

adapt its behavior to evade detection. Under certain conditions, the program may choose not to execute its malicious payload, while in others, it will carry out its intended behavior.

Malicious actors can also extract sensitive information from programs using dynamic analysis. Recent research [28] has proposed solutions to counter dynamic analysis methods (such as identifying encrypted keys in memory or uncovering the true control flow of a program).

2.2.3 Symbolic execution

Symbolic analysis aims to understand the behavior of a program by using symbolic variables instead of concrete values. To achieve this, input values are replaced with abstract symbols, such as a_1 in the code below. These symbolic inputs are then propagated through the code to explore multiple execution paths, helping to identify bugs, edge cases or malicious behavior. As the symbolic execution progresses, it generates logical constraints that must be satisfied for each path to be feasible. Tools like the Z3 solver [12] are commonly used to solve these constraints. Z3 is a constraint solver that takes a set of symbolic expressions as input and attempts to determine whether there exists a concrete assignment of values to the variables that satisfies all the constraints.

Based on a simple representation introduced by James C. King [17], we can illustrate symbolic execution using a basic function written in C, shown in Code 2.1. This function returns the result of operations performed on inputs A, B, and C. This allows us to compare its behavior under both concrete (Figure 2.1) and symbolic (Figure 2.2) execution. In both cases, the state of the program is shown after each statement. In concrete execution, variables are assigned specific values. In symbolic execution, variables are assigned symbolic expressions, each with associated constraints. For example, at statement 3:

- **Concrete** (Figure 2.1): variable `Y` is set to 8.
- **Symbolic** (Figure 2.2): variable `Y` is set to a_2 + a_3.

This example demonstrates that, in symbolic execution, values are not fixed. If this symbolic expression is later used in a condition, it can be resolved as an equation by a solver such as Z3.

```
int add(int A, int B, int C){
    int X = A + B;
    int Y = B + C;
    int Z = X + Y - B;
    return Z;
}
```

Code 2.1: Example C function for symbolic execution [17]

After statement	X	Y	Z	A	B	C
1	?	?	?	1	3	5
2	4	_	_	_	_	_
3	_	8	_	_	_	_
4	_	_	9	_	_	_
5	_	_	Returns 9	_	_	_

Figure 2.1: Concrete execution [17]

After statement	X	Y	Z	A	B	C
1	?	?	?	α_1	α_2	α_3
2	α_1+α_2	_	_	_	_	_
3	_	α_2+α_3	_	_	_	_
4	_	_	α_1+α_2+α_3	_	_	_
5	_	_	Returns α_1+α_2+α_3	_	_	_

Figure 2.2: Symbolic execution [17]

Another example to illustrate symbolic execution is presented as an execution tree. Below, the pseudo-code (Code 2.2) and its corresponding tree representation are shown in Figure 2.3, providing a clearer visualization of the different execution paths the program can take.

```
X = a_1
Y = a_2
if (X == 10){
    if (Y >= 0){
        return X;
    }
} else {
    if (Y > 0){
        return X;
    } else {
        return 0;
    }
}
```

Code 2.2: Example of conditional branching used to illustrate symbolic execution tree

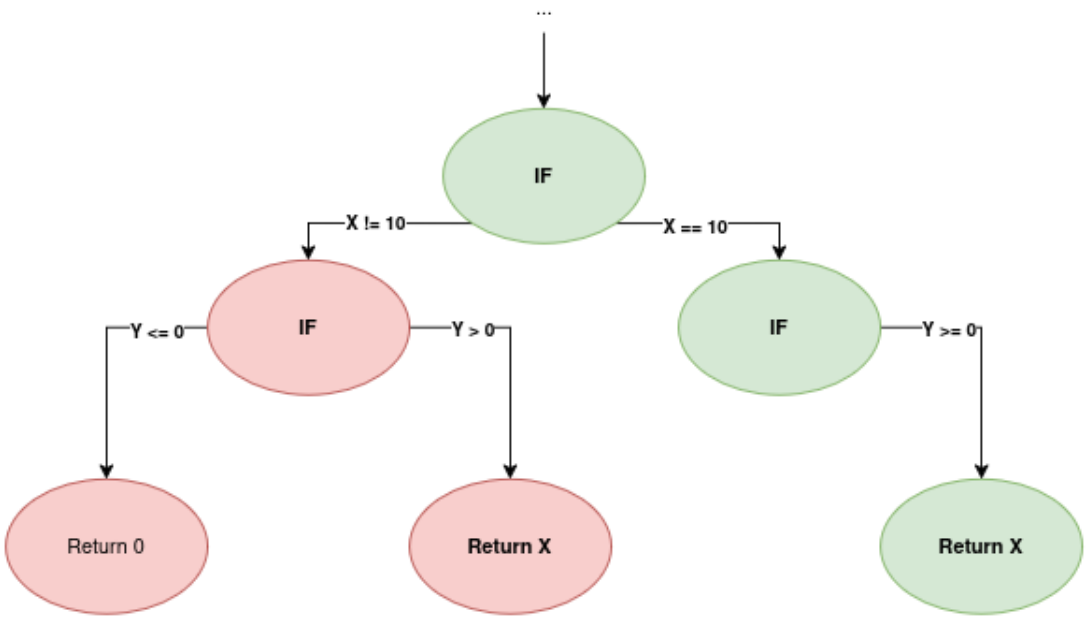

Figure 2.3: Symbolic execution tree

The Figure 2.3 represents tree used to model a control flow based on two symbolic variables, X and Y. This kind of structure is particularly useful in the context of symbolic execution, where instead of executing code with actual values, symbolic inputs are used to explore all feasible execution paths.

At the root of the tree, the decision depends on whether $X = 10$ or not. The execution continues to a second conditional based on Y. For instance, the path

leading to green `Return X` (on the right path) is only feasible if the symbolic execution engine can satisfy the constraint:

$$X = 10 \wedge Y >= 0$$

In the end, the combination of the constraints required to reach a node must be resolved by a solver. This process allows symbolic execution to identify potential bugs, or ensure complete path coverage. In the case of Figure 2.3, we can take the green path as an example. Solving its associated constraint allows us to verify that the return value is X.

2.2.4 Concolic execution

Concolic analysis combines **conc**rete execution (with real input values) and symb**olic** execution (with symbolic variables) [7]. It allows high path coverage and the flexibility to choose which parts of the code are executed concretely or symbolically [22]. This makes it possible to focus symbolic analysis on specific sections of the program instead of analyzing the entire program.

This technique can also be used to skip tasks such as packing (see Section 2.6). That is precisely how it will be used in this work. In the context of analyzing packed malware, symbolic execution can spend a significant amount of time in the unpacking phase. Concolic execution allows us to perform unpacking concretely, and then switch to symbolic execution once the original code is reached [15]. Moreover, concolic analysis helps detect malicious behaviors that might be missed due to evasion techniques.

There are several uses for concolic execution (see Section 2.7). In "Hybrid Concolic Testing" [22], the technique is used to solve constraints using Z3 in order to obtain concrete input values, rather than working only with symbolic constraints, helping to overcome some of the limitations of constraint solvers.

One of the main challenges in the concolic execution approach used in this work (starting with concrete execution followed by a transition to symbolic execution) is maintaining a memory map of the program state, to facilitate the switch to symbolic execution [15].

2.3 ANGR

SEMA is primarily based on angr, so it is essential to understand how angr works. Since this project mainly involves implementing and studying a specific feature within SEMA, it is crucial to grasp the functioning of the underlying tools it relies on. Angr is an open-source binary analysis platform for Python. It combines both static and dynamic symbolic (concolic) analysis, and provides a wide range of tools

for solving various binary analysis tasks [30]. In angr, there are 2 main points that is important to understand to have a clear view of SEMA, the path and the symbols.

The angr lifecycle (Figure 2.4) involves several core components working together to enable symbolic execution of binaries. Based on information found in the angr blog [3], Figure 2.4 can be described as follows. First, the CLE loader (green) loads the binary and determines its format and architecture, producing a memory map (as mentioned in 2.2.4). Then, Archinfo (brown) identifies architecture-specific details such as registers. PyVEX (blue), make it easier to analyze. SimEngine (orange) executes this representation symbolically by simulating program states and generating possible successors, while Claripy (grey) manages symbolic expressions and constraints using an SMT solver (Z3). SimOS (pink) provides high-level OS abstractions and system calls through symbolic summaries called SimProcedures. All of these components are combined in an angr project, which can then be used for tasks like path exploration.

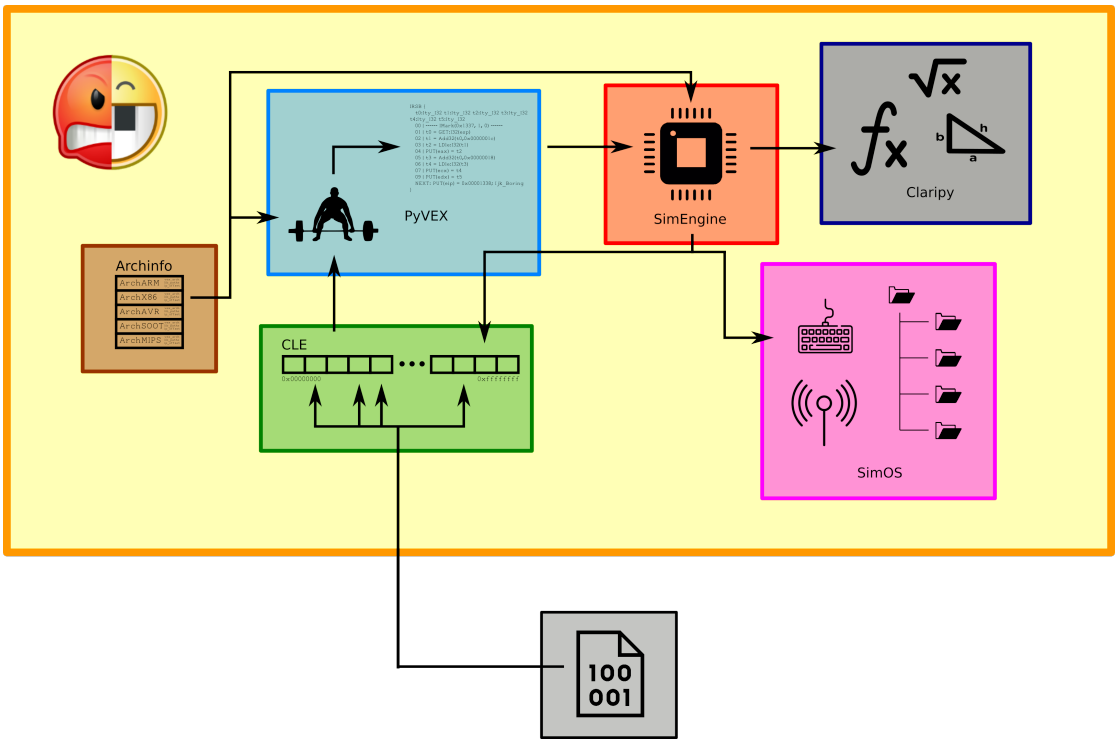

Figure 2.4: The angr lifecycle from angr's blog [3]

2.3.1 Execution path

The execution path in angr is represented as a succesion of SimState. A SimState is a state at one point of the execution and keep in memory a lot of information (register, constraint, etc). In our work, It is important to clearly understand how an angr state works within SEMA, as Symbion is based on the creation of new concrete states and the transition between these different states. Once the project is created, we can generate an initial state often at the entry point of the program. After the state is created, if we perform symbolic exploration from that point, it will eventually reach decision points that cause new states to be spawned (active states). Once a path of execution reaches its end, the state enters a deadend mode, meaning it has no further paths to explore.

2.3.2 Symbolic Expressions

Angr supports the creation and manipulation of symbolic variables through its constraint-solving engine [1]. These variables are internally represented as bitvectors, which are fixed-size sequences of bits capable of symbolically representing multiple possible values rather than a single concrete one.

Symbolic bitvectors can be created automatically by angr when symbolic input is required, typically via built-in mechanisms called SimProcedures (Section **??**). However, when automatic injection is insufficient or unavailable, users can manually define symbolic variables using Claripy, as demonstrated in Symbion's approach (Section 3.1.1). Creating a symbolic 64-bit variable and defining constraints (from [1]):

```
>>> x = claripy.BVS("x", 64)
>>> y = claripy.BVS("y", 64)
>>> state.solver.add(x > y)
>>> state.solver.add(y > 2)
>>> state.solver.add(x < 10)
>>> state.solver.eval(x)
4
```

2.3.3 Hooks and SimProcedures

SimProcedures in angr are designed to modify a program's behavior by hooking over specific functions or addresses in the program. They are quite powerful and can be used for a variety of purposes, including bypassing problematic functions or injecting custom behaviors during symbolic execution [2].

In angr, hooks are used to replace the execution of a function or a specific address in the binary with a custom behavior. Rather than running the original instructions, angr executes a SimProcedure. This technique is particularly useful for handling standard library calls, modeling complex behavior, or skipping parts of the code that are not essential to the analysis. Some studies, such as [21], focus on integrating SimProcedures into SEMA to better target specific malware families.

2.4 SEMA: Symbolic Execution-based Malware Analysis

SEMA (Symbolic Execution-based Malware Analysis) is an open-source tool designed for the automatic classification of malware using symbolic execution and based on angr. It relies on a structured toolchain that analyzes a binary, extracts a behavioral representation in the form of a graph, and compares this representation to known malware families.

2.4.1 Toolchain architecture

The global functioning of SEMA [6] can be summarized in the following steps (Figure 2.5) :

1. Binaries we want to analyze
2. Each binary is analyzed using angr, a symbolic execution framework under certain parameters such as exploration techniques, timeouts, and others. Angr symbolically executes the binary by replacing concrete input values with symbolic variables. This enables exploration of multiple execution paths and the capture of key behavioral elements, such as system calls and their arguments.
3. The execution traces are then combined to construct a System Call Dependency Graph (SCDG), where each node corresponds to a system call, and edges represent dependencies.
4. SCDGs from binaries of the same family are mined using graph pattern discovery techniques to extract subgraphs and associated signatures for each family.
5. To classify a new binary, its corresponding SCDG is generated and compared to the known signatures using similarity.

2.4.2 Enhancing SEMA

Before diving into the improvements proposed in this work, it is important to understand the architecture of the current version of SEMA (without red component). The following diagram provides an overview of the toolchain, from symbolic execution to malware classification using SCDG graphs :

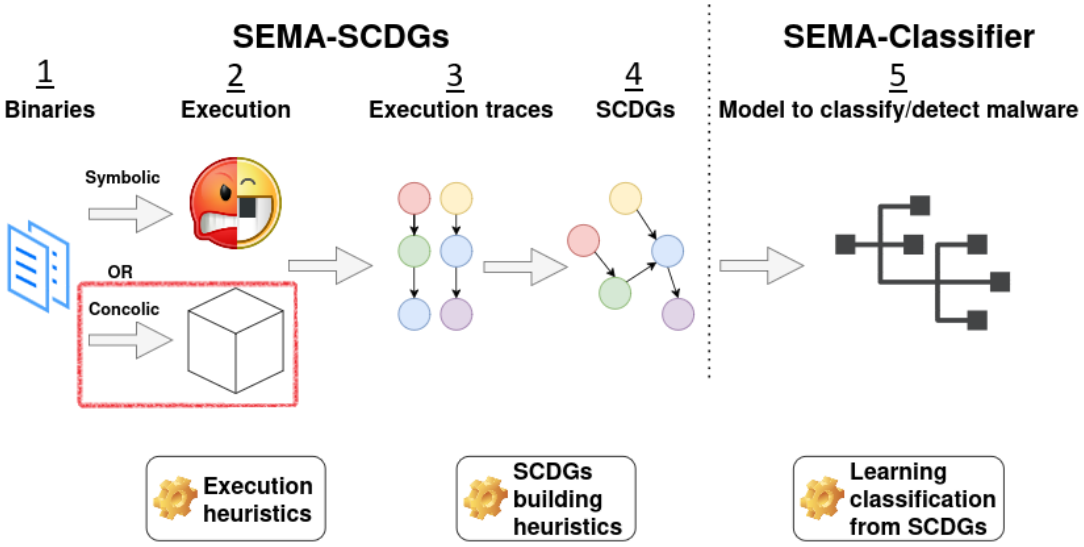

Figure 2.5: Enhancing SEMA with concolic execution[10]

Now that we have a better understanding of the main tool used in this work, we can focus on the specific component that concerns us. The SEMA-SCDGs module is responsible for the analysis and symbolic execution of the binary, and it is precisely this component that we will examine in detail. It is clear that, to support concolic execution, we will need to focus on the section labeled "Symbolic Execution" in the architecture diagram. The main objective of this thesis is to append the red component in Figure 2.5.

2.5 Symbion

Symbion [15] is a new tool integrated into the symbolic execution framework angr, designed to bridge the gap between concrete and symbolic execution. It addresses the limitations of purely symbolic analysis, by enabling analysts to concretely execute a program up to a certain point, switch into symbolic execution to control flow or compute input values, and then resume concrete execution with the symbolic results applied (or vice-versa). This hybrid approach is especially useful when

dealing with complex binaries, where full symbolic execution is infeasible, such a packed file.

As explained in "Interleaving Symbolic with Concrete Execution" [15], this approach enables the interleaving of symbolic and concrete execution. More specifically, interleaved concolic execution is performed in three main (only two of these phases are summarized in this work, the two that are used) phases described in Symbion's paper as :

- **Phase 1: Initial Concrete Execution :** An entry point is always required — here referred to as the Concrete Starting Point (CSP) — from which the binary is concretely executed until it reaches a Point of Interest (PoI). As described in Section 2.3.1, at the PoI a new state is created, and both memory and register values are saved.
- **Phase 2: Interleaved Symbolic Execution :** Using the symbolic engine (Figure 2.4) the state is synchronized. As mentioned earlier, the memory and register values are reused, and part of the state can be set as symbolic variables.

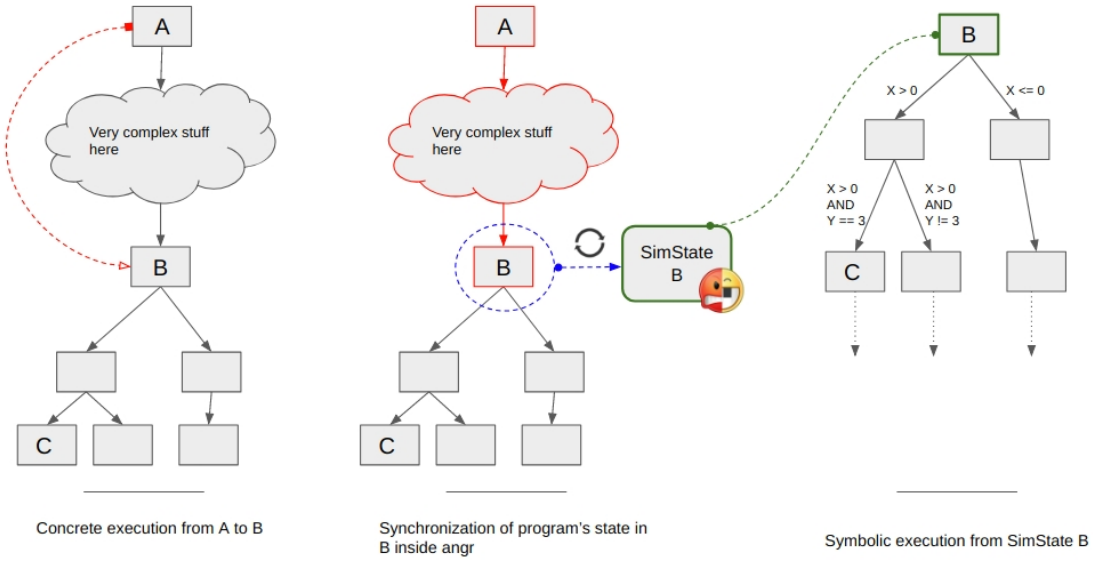

Figure 2.6: Symbion workflow from Angr's blog[33]

For example, in Figure 2.6, the different cells represent the states during execution. At the root, the component A corresponds to the Concrete Starting Point (CSP). Then, we begin the concrete execution of our binary until reaching a

Point of Interest (PoI), labeled B. This step is used to bypass code that is complex to analyze symbolically, such as a decompression routine in a packed binary. The various calls or SimProcedures during this phase are not of interest to us. Once we reach state B, we need to synchronize the concrete state B with a symbolic state B (on the right side of Figure 2.6). At this point, the memory information is synchronized, and certain parts of the state can be set as symbolic portions. An example (Figure 2.7) of such a transition can be found in "Interleaving Symbolic with Concrete Execution" [15] :

Concrete State	**Symbolic State**
`mem[W] = 0`	`mem[W] = 0`
`mem[X] = 1`	`mem[X] = sym_var_A`
`mem[Y] = 5`	`mem[Y] = sym_var_B`
`mem[Z] = 7`	`mem[Z] = 7`

Figure 2.7: Comparison between concrete and symbolic memory states at Point B [15]

Finally, we can resume symbolic analysis from our new state B (shown in green in Figure 2.6).

2.6 Packing

An executable can exist in different formats, such as ELF for Linux or PE for Windows. The ELF format was discussed in detail in Section 2.1.1. The primary purpose of a packer is to reduce the file size, encrypt the contents, and protect the data from analysis or reverse engineering [35].

2.6.1 The packer problem

Packing is commonly used by malicious actors to obfuscate code. As noted in the UPX documentation, the packed file is “completely self-contained and runs exactly as before”. This characteristic allows malicious actors to preserve a fully functional executable while hiding as much information as possible, making both static and dynamic analysis significantly more difficult.

In addition, attackers can modify the code of a packer to evade detection by traditional tools (Table 2.1). This customization turns a known packer into an unknown variant, presenting a major challenge for malware analysts. Further-

more, attackers can combine multiple packers in layers, leading to complex packer detection.

2.6.2 UPX

The Ultimate Packer for eXecutables [25] (UPX) is an open-source, cross-platform executable packer. It is based on a fast compression algorithm, although users can add their own algorithms if needed. UPX claims to outperform traditional tools like Zip, offering extremely fast decompression speeds (up to 500 MB/s). For example, in this study, the unpacked binary `tests/x86_64/packed_elf64` [7] was 929 kB, compared to only 407 kB after being packed using UPX [25] (version 3.91).

In addition, UPX remains actively developed and maintained. Once a file is packed with UPX, it loses a significant amount of information that is useful for static or dynamic analysis. Some of this information is only restored during the runtime decompression phase [16].

2.7 Literature Review

One of the first challenges in research on concolic execution is the terminology itself. The word "concolic" is a combination of "concrete" and "symbolic". However, the term is not always used consistently in the literature. It may also appear as "hybrid execution" [37] or even under different names depending on the application domain. Since concolic execution is not exclusive to malware analysis, it is also found in other fields such as software testing [22], where it may be referred to as "hybrid testing" or "concolic testing."

Due to its hybrid nature, which combines two distinct techniques, concolic execution can be implemented in various ways. For example, an analysis may begin with concrete execution and switch to symbolic execution when necessary, or vice versa. Both modes are supported within Symbion [15]. In other cases, the concrete phase may be enhanced using a fuzzer [22], leading to more efficient path exploration. Alternatively, as noted in "Concolic Testing" [29], both concrete and symbolic executions can “run simultaneously, and each gets feedback from the other”.

These variations highlight the flexibility of the concolic execution, but they also contribute to inconsistencies in terminology. Furthermore, this area of research remains highly active, with ongoing exploration of alternative analysis methods, for example, converting binary files into grayscale images [23].

SEMA does not currently support concolic execution. It will likely be capable of it in the near future. In the meantime, a list of relevant tools that support concolic

[7] `https://github.com/angr/binaries`

execution or serve as a foundation for concolic extensions can be found in Table 2.1.

Tool	**Refs**	**GitHub**	**Last Update**	**Descriptions**
oss-sydr-fuzz	[34]	ispras/oss-sydr-fuzz	May 28, 2025	sydr-fuzz that combines fuzzing (libFuzzer, AFL++) with the power of dynamic symbolic execution.
ANGR	[30]	angr	May 27, 2025	A powerful and user-friendly binary analysis platform.
Driller	[32]	shellphish/driller	Mar 24, 2025	Driller was built on top of AFL with angr being used as a symbolic tracer.
KLEE	[8]	klee/klee	May 2, 2025	KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure.
Triton	[27]	JonathanSalwan/Triton	Feb 15, 2025	Triton is a dynamic binary analysis library.
Binsec	[11] [5]	binsec	Feb 17, 2025	Open-source toolset to help improve software security at the binary level.
S2E	[9]	S2E	Dec 7, 2024	A platform for multi-path program analysis with selective symbolic execution.
Manticore	[24]	trailofbits/manticore	Jul 11, 2023	Manticore is a symbolic execution tool for the analysis of smart contracts and binaries.

Table 2.1: Overview of symbolic and concolic analysis tools (descriptions taken directly from the respective GitHub pages).

Chapter 3

Design and Implementation

Based on the literature review (Section 2.7) and discussions with the team supervising my master's thesis, the decision was made to use Symbion, the concolic execution engine integrated into angr, as a suitable solution for extending SEMA. This choice was guided by the intention to avoid introducing an entirely new tool and instead to leverage the components already provided by the angr framework. However, this decision is not without challenges: Symbion is no longer actively maintained[1], and one of its key dependencies, angr-targets, was archived by its maintainers on January 7, 2025[2].

Despite these limitations, the goal of this thesis was clearly defined: to make Symbion work within the existing SEMA architecture. As discussed earlier, concolic execution can be applied in different ways. In this work, the selected approach is running the program concretely up to a specific point (usually after the unpacking phase) then switching to symbolic execution to carry out the analysis.

In this chapter, we will first study and reproduce the example described in the Symbion documentation [33] (Section 3.1). Then, in Section 3.2, we will apply the same methodology to different custom samples designed for validation purposes. Finally, in Section 3.3, we will build on the insights from the previous sections to integrate Symbion into SEMA.

3.1 Reference Examples

In this section, we analyze the example presented in the Symbion documentation [33]. Our objective is to reproduce and understand, step by step, the analysis of the file `packed_elf64`, which can be found at the following repository: `https://github.com/angr/binaries.git`.

[1] https://github.com/angr/angr/issues/2701
[2] https://github.com/angr/angr-targets

3.1.1 Static analyses

In the following section, we begin by performing preliminary static analysis on the binary. As a first step, we execute the file and observe its behavior:

- [+] **Parsing malware configuration**
- [+] **Virtual environment detected!**

By analyzing the binary using techniques presented later in Chapter 4, we can observe that it is packed with UPX version 3.91. The fact that the binary is packed confirms that this is a relevant case for applying concolic execution, allowing us to handle the unpacking phase concretely and then switch to symbolic execution (see Section 2.2.4). Let us analyze the unpacked binary using IDA Pro (`https://hex-rays.com/ida-pro`):

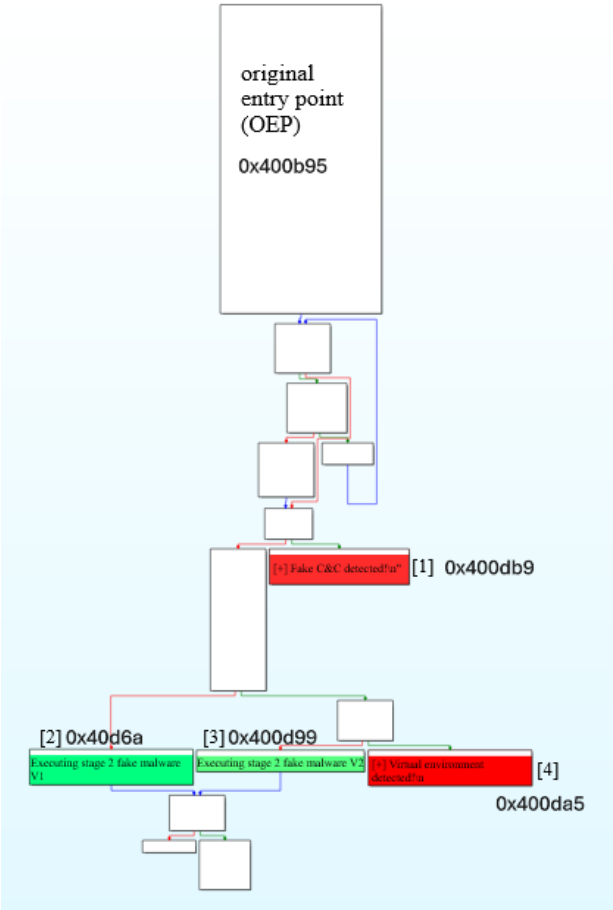

Figure 3.1: Overview of the Binary's Control Flow Graph in IDA Pro

In Figure 3.1, the red section represents the evasion techniques, while the green section highlights the malicious behavior. Below is additional information about Figure 3.1:

Block	Behavior	Block Start Address
1	stdout : Fake C&C detected	0x400DB9
2	stdout : Executing stage 2 fake malware V1	0x400D6A
3	stdout : Executing stage 2 fake malware V2	0x400D99
4	stdout : Virtual environment detected	0x400DA5

Table 3.1: Legend for Figure 3.1: Behavioral Blocks in the Control Flow Graph

If we look more closely at the relevant section in Figure 3.2, we can identify the decision point highlighted in green and address, found using IDA Pro, is `0x400cd6`. Finally, the four different behaviors are shown in blue on the diagram.

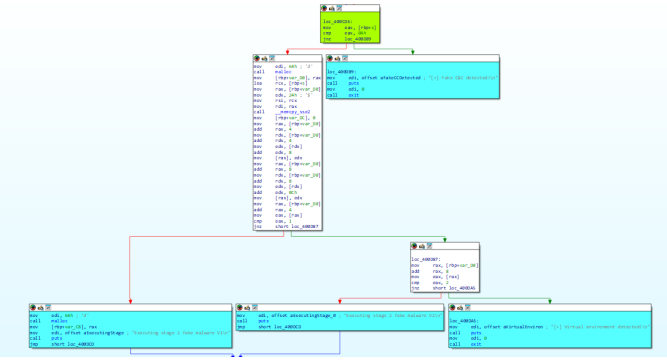

Figure 3.2: Overview of the Binary's Control Flow Graph in IDA Pro (zoom)

3.1.2 Concolic execution

Now that we have identified the relevant addresses, we can begin the analysis of the packed binary. As previously mentioned, we will follow the methodology from the Symbion example [33]. To enable the interaction between symbolic and concrete execution, Symbion relies on two key components: angr-targets[3] and avatar2[4]. The angr-targets library provides implementations of ConcreteTarget objects, which serve as interfaces to control concrete execution environments (GDB, QEMU, hardware debuggers). These targets expose standardized APIs for reading/writing memory, accessing registers, setting breakpoints, and controlling execution. Avatar2

[3]https://github.com/angr/angr-targets
[4]https://github.com/avatartwo/avatar2

is used as the orchestration layer; such as synchronization between angr's symbolic engine and the external concrete target.

First, to perform concolic analysis, we need to configure a concrete execution target using avatar2 and angr-targets. This is typically done by launching a GDB server and instantiating a concrete target, as illustrated in the code snippet below:

```
# Spawning of the gdbserver analysis environment
print("gdbserver %s:%s %s" % (GDB_SERVER_IP,GDB_SERVER_PORT,
    binary_x64))
subprocess.Popen("gdbserver %s:%s %s" % (GDB_SERVER_IP,
    GDB_SERVER_PORT,binary_x64),
                 shell=True)

# Instantiation of the AvatarGDBConcreteTarget
avatar_gdb = AvatarGDBConcreteTarget(avatar2.archs.x86.X86_64,
                                     GDB_SERVER_IP,
    GDB_SERVER_PORT)
```

The version above has been slightly modified from the original, and arguments such as `stdout=subprocess.PIPE` have been removed to allow more visible feedback during execution. Based on the addresses identified in previous section, we can initialize the following variables:

```
STAGE2_V2 = 0x400d99
STAGE2_V1 = 0x400d6a
VENV_DETECTED = 0x400da5
FAKE_CC = 0x400db9
BINARY_EXECUTION_END = 0x400def
```

At this point, the initialization of a Symbion project is done as shown in the code below. Later in this work, we will encounter a challenge related to deciding whether to create a concrete or symbolic project.

```
# Creation of the project with the new attributes 'concrete_target
    '
p = angr.Project(binary_x64, concrete_target=avatar_gdb,
                              use_sim_procedures=True)

entry_state = p.factory.entry_state()
entry_state.options.add(angr.options.SYMBION_SYNC_CLE)
entry_state.options.add(angr.options.SYMBION_KEEP_STUBS_ON_SYNC)

simgr = p.factory.simgr(state)
```

The variable p (project) is created using the concrete_target argument, which connects angr to the GDB server for concrete execution. The following Figure 3.3 from "SYMBION: Interleaving Symbolic with Concrete Execution" [15] illustrate the Symbion architecture :

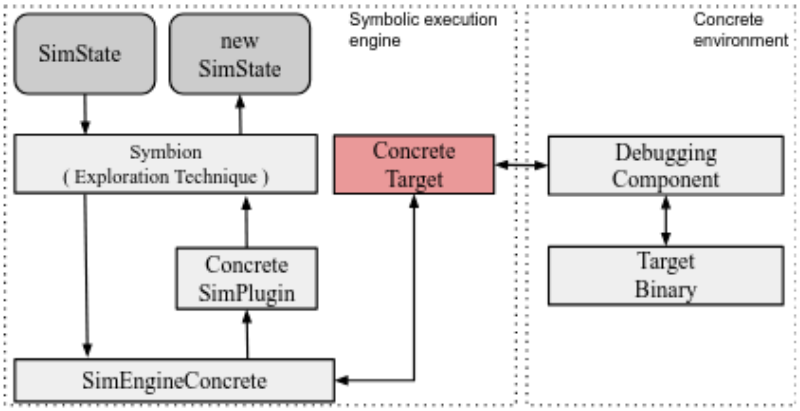

Figure 3.3: Overview of Symbion architecture.

The red part in the figure is the main part of Symbion with SimEngineConcrete. The Concrete Target is the module that allows the connection to, for example, a GDB server. During the first pass (from the SimState to the Concrete Target), the SimEngine executes the target binary and adjusts registers, memory, etc. During the second pass, the SimPlugin converts the concrete (real) state into a symbolic state (new SimState) (see Section 2.5).

Then, the Symbion exploration technique allows us to set a PoI before switching back to symbolic execution. As shown in the following lines of code:

```
# Explore the program concretely until we reach 0x400cd6
simgr.use_technique(angr.exploration_techniques.Symbion(find=[0
    x400cd6]))
exploration = simgr.run()
new_concrete_state = exploration.stashes['found'][0]
```

Angr explores the binary concretely (real execution) to reach the address **0x400cd6**. If angr finds a matching address (reaches the PoI), it will generate a new state, as shown in Figure 2.6. At this point, we can reuse this new state to explore the binary symbolically. As mentionned in section 2.3.2 and 2.5. We simply need to specify which memory segment we want to make symbolic. By examining our decision point (Figure 3.2) using IDA Pro or GDB, we observe that this address is `rbp-0xc0`.

```
# Declaring a symbolic buffer
arg0 = claripy.BVS('arg0', 8*32)

# The address of the symbolic buffer would be the one of the
# hardcoded malware configuration
symbolic_buffer_address = new_concrete_state.regs.rbp-0xc0

# Setting the symbolic buffer in memory!
new_concrete_state.memory.store(symbolic_buffer_address, arg0)
```

In theory, this represents a high-level approach that can be used to analyze our malware. However, in practice, there are some limitations that arise during these different steps, which will introduce constraints and limitations in this work. One of these challenges is that the binary is packed. We will explain in more detail why this is a problem and discuss possible approaches to address some of these limitations in chapter 4 (Packing).

The main issue with packed binaries is that during execution (specifically during the unpacking phase), breakpoints set at key locations such as the OEP (`0x40b95`) or the PoI (`0x400cd6`) cannot be reached, as the corresponding code is still being mapped into memory, meaning those addresses do not yet exist or the breakpoints may be overwritten or rendered ineffective.

For now, the Symbion example [33] provides a straightforward solution for handling packed binaries (code below). By using Symbion's exploration technique described earlier, we can trigger the end of the unpacking phase and then gain access to the actual addresses of PoI. Strategies to identify this address are further discussed in Chapter 4 (Packing). Here, we can assume that the address of interest is `0x85b853`. When this address is reached, a new stub becomes available at `0x45b97f`, and the unpacking stub then enters a loop, performing tasks such as copying chunks of code into memory. Through manual reverse engineering, the author of the Symbion documentation identifies `0x45b97f` as a key transition point between unpacking stages.

```
simgr.use_technique(angr.exploration_techniques.Symbion(find=[0
    x85b853]))
exploration = simgr.run()
new_concrete_state = exploration.stashes['found'][0]

# Hit the new stub 4 times before having our unpacked code at 0
    x400cd6
for i in xrange(0,4):
    simgr = p.factory.simgr(new_concrete_state)
    simgr.use_technique(angr.exploration_techniques.Symbion(find
    =[0x85b853]))
    exploration = simgr.run()
    new_concrete_state = exploration.stashes['found'][0]
```

Finally, we have reached our Point of Interest (PoI : `0x400cd6`), and the decision address has been made symbolic. From this point, we can take the newly created state and perform a classic symbolic exploration and targeting, for example, behavior 3 as illustrated in Figure 3.1. It is also possible to explicitly avoid certain addresses during the exploration, allowing us to guide the analysis toward a specific path. This results in a unique execution trace, and once the target address is reached, we can run the concrete process (simgr) and observe the program's output directly through the GDB server.

```
# Symbolically explore the malware to find a specific behavior by
    avoiding
# evasive behaviors
exploration = simgr.explore(find=DROP_STAGE2_V2, avoid=[
    DROP_STAGE2_V1,

    VENV_DETECTED, FAKE_CC ])
```

By testing the different cases, we can observe the following outputs:

- Find V1 and avoid V2/Fake CC :

```
[+]Parsing malware configuration
Executing stage 2 fake malware V1
```

- Find V2 and avoid V1/Fake CC :

```
[+]Parsing malware configuration
Executing stage 2 fake malware V2
```

3.2 Validation of the Methodology

In this section, we aim to validate the methodology described in the Symbion documentation [33]. To do so, we create several custom samples designed to test different scenarios. The sample creation process follows a straightforward logic, as illustrated in Figure 3.4. The goal was to design a code template with a critical section (symbolic analysis) and a malicious behavior component, an "evil" function.

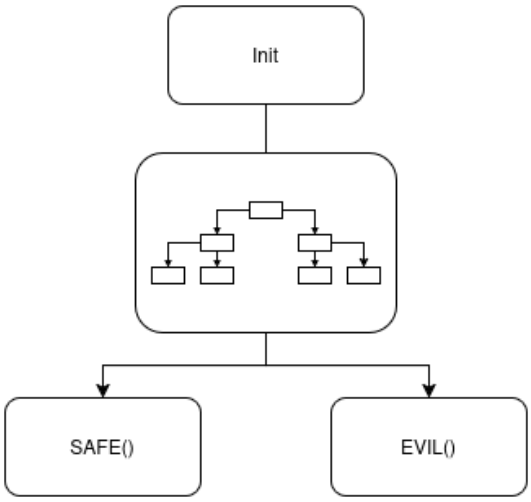

Figure 3.4: Creation of sample.

Although simple, the different files are used to test various behaviors and to allow for multiple configurations.

- **Testing feasibility:** To check when we can apply the methodology from the reference example to one of our sample (simulating malicious behavior).
- **Testing limitations:** To intentionally break the methodology and observe the limitations of the current Symbion implementation.
- **Testing different scenarios:** This allows us to test multiple scenarios with different configurations, such as simulating a routine before the start of the important code (simulated by printf statements).

All binaries created in this work were compiled using the following options :

- Compiler : **gcc** (the GNU compiler).
- Option 1 :**-no-pie** (to disable position-independent executable (PIE)).
- Option 2 :**-fno-pie** (to disable generation of position-independent code).

Option `-fno-pie` disables the generation of position-independent code at compile time, while `-no-pie` tells the linker not to produce a PIE executable, both are required to ensure the binary is fully non-PIE[5]. It ensures the generated binary has fixed addresses, which simplifies reverse engineering and symbolic/concolic analysis.

[5]https://stackoverflow.com/questions/74300190/difference-between-fno-pie-and-no-pie

The listing presents different samples (below you can find the code of `sample_01`):

- **Sample01:** This first example serves as a starting point to test our program and use Symbion.
- **Sample02:** This second example produces output during concrete execution in console. It allows testing different target addresses for concrete execution. One can try placing a PoI on the second `print` to confirm that the variable `X` has already been initialized.
- **Sample03:** This final example will be more useful later on, but already allows us to verify the same situations as before, with a variable that is not fixed as in the previous samples.

```c
# Sample 01
#include <stdio.h>

char* evil() {
    return "I'm EVIL !\n";  // Malicious behavior
}

char* safe() {
    return "I'm safe !\n";  // safe function (represents evasion)
}

int main(int argc, char *argv[]) {
    int x = 10;   // Address: rbp-0x4 (symbolic memory)
    if (argv[1] != NULL) {
        x = atoi(argv[1]);
    }
    if (x > 0) {
        if (x % 3 == 0) {
            if (x % 5 == 0) {
                printf("%s", evil());
                return -1;
            }else{
                printf("%s", safe());
                return 1;
            }
        }else{
            printf("%s", safe());
            return 2;
        }
    }else{
        printf("%s", safe());
        return 3;
    }

    return 0;
}
```

At this stage, the binaries are not packed and serve to validate the previously described methodology. We can now apply this methodology, which can be summarized as follows:

1. Start the GDB server and create a new angr project.
2. Execute the binary concretely using the Symbion exploration technique, up to the target address (PoI) (see Table 3.2).
3. Create the new concrete state (see Figure 3.3).

4. Launch symbolic exploration to discover the malicious behavior, while avoiding evasion techniques (represented here as safe calls we want to bypass).

5. Once the path is found, execute it concretely to display the output in stdout "I'm EVIL!" message.

Let us now analyze our samples in IDA Pro to retrieve the relevant addresses (Table 3.2) and observe the different execution paths (Figure 3.5).

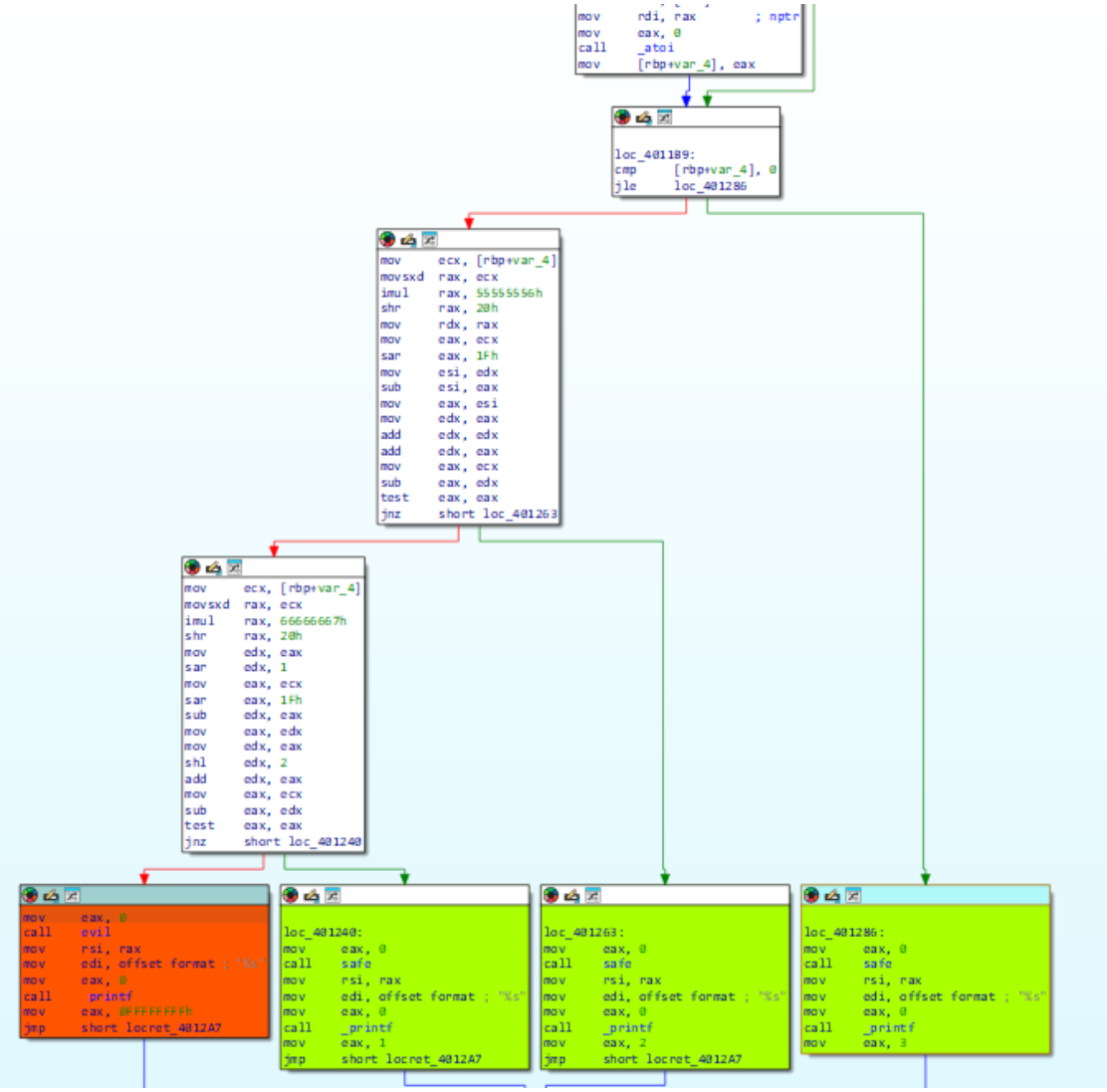

Figure 3.5: Control Flow Graph of tfe_sample01 Generated with IDA Pro

Targets	tfe_sample01	tfe_sample02	tfe_sample03
Concrete target	0x4011b9	0x4011f7	0x40119c
Drop safe behavior	0x401165	0x401185	0x401165
Malicious behavior	0x401156	0x401176	0x401156
Binary END	0x4012a8	0x4012e6	0x4012ab

Table 3.2: Concrete Execution and Behavioral Addresses for tfe Samples

By executing the binary `tfe_sample01` (available in `sema_toolchain/database/Binaries/thomas`), you will observe the next output. This sample serves as a test case to validate that the symbolic execution environment is correctly set up and that the integration with Symbion can be functional.

```
root@f5e20fb548e3:/sema-scdg/application# ./database/Binaries/thomas/tfe_sample0
I'm safe !
```

Figure 3.6: Console Output from Executing tfe_sample01

If we set the addresses from Table 3.2 and apply the methodology, the following output is observed. We found three states in the `avoid` stash and one state in the `found` stash. This confirms that we successfully avoided the three addresses provided in the `avoid` argument of angr's exploration function, while reaching the one we were targeting (the `evil` function). At the end of the execution, since we perform a `run` on the simgr object, the GDB server output displays: `"I'm EVIL!"`, which confirms that the program was successfully executed using Symbion.

```
Reach concretly the first decision point
Symbolically executing binary to find the address of the 'EVIL'
    function : 0x401156
FIND :
  - active: 0
  - stashed: 0
  - pruned: 0
  - unsat: 0
  - errored: 0
  - deadended: 0
  - unconstrained: 0
  - found: 1
  - avoid: 3
Malicious behavior found!
Executing binary concretely
I'm EVIL !
```

A critical point to understand in this example is the role played by the structure of our binary. Conditional branching is a key element of success, as the control flow

is precisely what makes concolic reasoning valuable in this context. Indeed, the address we render symbolic after the concrete to symbolic transition is a crucial element in each condition. Do not forget that a symbolic buffer is created to replace the value of `x`. Now that we are discussing our symbolic segment, let us examine how it was determined, inspect how variables are initialized, and determine the most appropriate Point of Interest (PoI) and buffer for symbolic execution.

```
0x000000000040119c <+40>:    je     0x4011b9 <main+69>
0x000000000040119e <+42>:    mov    -0x20(%rbp),%rax
0x00000000004011a2 <+46>:    add    $0x8,%rax
0x00000000004011a6 <+50>:    mov    (%rax),%rax
0x00000000004011a9 <+53>:    mov    %rax,%rdi
0x00000000004011ac <+56>:    mov    $0x0,%eax
0x00000000004011b1 <+61>:    callq  0x401060 <atoi@plt>
0x00000000004011b6 <+66>:    mov    %eax,-0x4(%rbp)
0x00000000004011b9 <+69>:    cmpl   $0x0,-0x4(%rbp)
```

It is important to understand that our code executes concretely up to address `0x4011b9`. At this point, the variable `X` has already been initialized (concretely) at address `rbp-0x4`. We must now use the commands provided by Symbion to make a portion of the memory symbolic, so that during symbolic execution, this variable can be adjusted to satisfy future constraints. To proceed, one must first define the symbolic buffer (arg0), specify the corresponding memory address intended for symbolic execution, and apply the symbolic mapping accordingly as seen before.

```
# Declaring a symbolic buffer
arg0 = claripy.BVS('arg0', 32)
symbolic_buffer_address = new_concrete_state.regs.rbp-0x4
new_concrete_state.memory.store(symbolic_buffer_address, arg0)
```

In this scenario, the Point of Interest (PoI) is positioned after the initialization of the variable `X`, at address `0x4011b9`. Consequently, attempting to define the PoI at an earlier address would render the symbolic memory setup ineffective, as the variable would not yet be initialized. In such a case, the execution reaches the `safe()` behavior, as the program's memory is initialized earlier, even though the value `10` has not yet been explicitly set.

3.3 Integration into SEMA

The first step in integrating Symbion into SEMA is to determine what is required to successfully run the reference example [33]. Based on Section 3.1.2 of this work and further explanations we can present the current version used to run Symbion inside SEMA in Table 3.3.

Name	Version SEMA	Version used
angr	9.2.21	8.20.7.27
archinfo	9.2.21	8.20.7.27
claripy	9.2.21	8.20.7.27
cle	9.2.21	8.20.7.27
pyvex	9.2.21	8.20.7.27
ailment	9.2.21	8.20.7.27

Table 3.3: Dependencies and Corresponding Versions

All the versions used are available on GitHub (`sema_toolchain/sema_scdg/requirements.txt`). These versions are used during the creation of the Docker image.

Before implementing, it is important to clearly understand how SEMA SCDG works and what the relationships are between its various methods (Figure 3.7) in order to place our code snippets in the right places. The SemaSCDG class creates an object based on a `.ini` file located in the configs folder. This file contains many parameters that can be configured to customize our symbolic execution (such exploration technique, max number execution, etc). The first step is to add new arguments to the `.ini` file:

- **symbion_on :** true or false to use concolic execution.
- **target :** the first target that is reached by concrete execution.
- **end_addr :** the end of the binary.
- **symbolic_memory :** part of memory that will become symbolic.

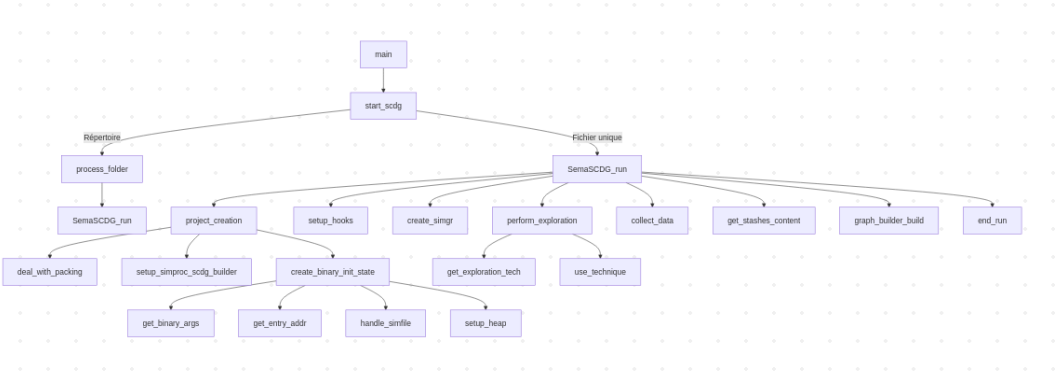

Figure 3.7: SemaSCDG flowchart

As we mentioned earlier, it is important to set up an angr project with the correct arguments. SEMA set the project depending on different parameters in the methods `project_creation` → `deal_with_packing` on Figure 3.7. Then we can catch the new argument `self.symbion_on` to configure the concrete project and launch the GDB server. In later discussions and implementations, this method will no longer return a single project but two (one using symbolic parameter and is similar to the one used previously in SEMA and one for concrete execution using the concrete target argument). We will discuss this implementation choice later. Furthermore, this is the method where the GDB server is spawned. Then, the last step is to configure the concrete state to reuse the SEMA tool as usual.

Finally, as mentioned above, the last step is to create our new concrete state. To achieve this, we can reuse the state already created by SEMA directly within the `create_binary_init_state` method show in Figure 3.7. This allows us to create a `concrete_state` by reaching the Point of Interest (PoI) through concrete execution. The new method `create_concrete_state` takes two arguments:

- **proj:** the project initialized for concrete execution.
- **state:** the initial state created by SEMA.

Although the implementation may seem simple, it is essential to fully understand how calls and SimProcedures are managed in SEMA, Symbion, and how they interact with each other. This solution, while appearing straightforward, was found after extensive testing and experimentation. Indeed, if we apply the same code but this time with a different configuration file, that targets sample_02 and its associated arguments (Figure 3.9), we obtain the results shown below from executing samples 1 and 2:

Figure 3.8: Execution Result of tfe_sample01 Binary

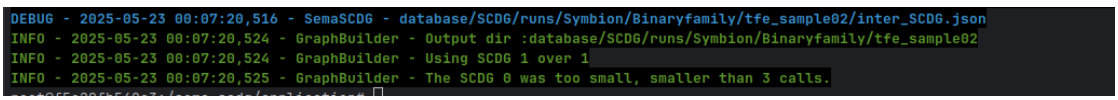

Figure 3.9: Execution Result of tfe_sample02 Binary

The first observation is that during execution, no system calls are observed. This is a critical issue, as the presence of system calls is essential for constructing

our graph. One validated explanation, confirmed through multiple tests, is that during the concrete execution phase (via Symbion or another concrete engine) leading up to the target address, some calls or SimProcedures (puts, printf, etc.) are already loaded concretely and their effects are resolved and stored as concrete values in memory. Once this happens, these functions are no longer recognized as symbolic calls when angr resumes symbolic execution (SEMA). However, an inelegant solution is to manually hook the printf that is no longer being recognized, and to add a parameter in the configuration file to enable a debug condition.

```
if self.symbion_on and self.symbion_debug:
            proj.hook(0x401070, angr.SIM_PROCEDURES['libc']['
    printf'](), length=5)
```

Having successfully executed our sample using Symbion, the last step is to try on the reference example (packed ELF binary). The reference sample has been successfully executed within SEMA. However, it does not yet account for all system calls. Addressing this limitation falls outside the scope of this work but is noted for future research. Ongoing efforts are focused on extending support for additional calls and SimProcedures (hooks). A block of code was added to our create_concrete_state method to handle the unpacking of the file. However, if we take a look inside final_SCDG.gv, we can see our different behaviors:

- **Fake C&C detected!**
- **Virtual environment detected!**
- **Executing stage 2 fake malware V1**
- **Executing stage 2 fake malware V2**

Concerning our sample, it works as well. By using a new parameter, we can add an option to trigger the execution loop to unpack the binary and thus reach the part we are interested in. A final parameter is added to obtain the address marking the end of the unpacking process. In the following chapter 4, we will discuss the techniques used to identify the appropriate target addresses. The method is relatively straightforward, the PoI can correspond to the address we aim to reach through concrete execution. In contrast, locating the address corresponding to the end of the unpacking process requires a more involved analysis, which will be addressed in the next chapter.

Chapter 4

Packing

This section focuses on packing. While packing was briefly addressed in the previous chapter, we will now take a closer look at its specific features and perform a more targeted analysis, particularly focusing on how to identify relevant addresses. We will specifically analyze how our binaries are packed. We will also carry out a series of analyses that can be performed before using SEMA. This preliminary analysis, based on either static or dynamic techniques, can help us better understand the analyzed files. Our analysis will focus solely on binaries packed with UPX.

4.1 Clarifying the Context

It is worth recalling where we currently stand: we have enabled SEMA to execute certain files using concolic execution. To achieve this, we concretely perform the unpacking phase, and SEMA uses this new concrete state as a starting point for its symbolic exploration. While there is still work to be done in adapting the analysis of the resulting graph and recovering elements lost during the transition from concrete to concolic execution, these aspects are beyond the scope of this study.

To launch execution successfully, we need to identify specific information such as key addresses, the type of packer (UPX), and its version. In order, we can follow a series of steps designed to maximize the chances of success when analyzing a new binary.

4.2 Static Analysis Techniques

To better understand the behavior of our binary, it is important to analyze it. Various tools exist for this purpose, whether for automatic or manual analysis [35]. In our case, SEMA is not yet capable of generating the information required for

our execution. When automated tools are not available, the program must be analyzed statically. To do so, many reverse engineering techniques and tools are available [31].

Let's start by examining how our program behaves in GDB. Once loaded into GDB, running `disass` main no longer produces output and displays the message: No symbol table is loaded. Use the "file" command. Using the `file` command returns: No executable file now/No symbol file now. This immediately shows that analyzing our original binary is no longer straightforward. This is precisely why packers are used in the context of malware, as mentioned in Section 1.1.

By using the simple command `strings file_name | grep -i upx`, we obtain the following output:

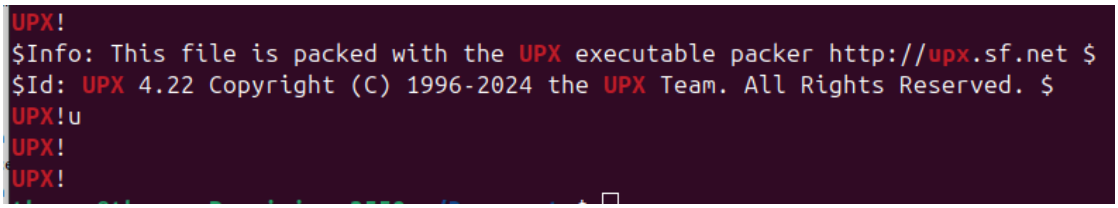

Figure 4.1: Screenshot of the `strings` command output

As shown in Figure 4.1, the version of UPX used to pack this binary is 4.22. We can also confirm that the file was indeed packed with UPX. However, it is worth noting that some individuals may use UPX as a base and slightly modify its code to make the detection of the packer type more difficult [13]. This information can also be retrieved by using the UPX option "upx -t".

Furthermore, we can continue our analysis using reverse engineering tools such as IDA Pro. As part of this work, we received a classroom license to assist us in the analysis of our malware samples. We have already used this tool, as shown in Figure 3.1. We will now use it again to analyze our binary in both its packed and unpacked forms. Our goal is to manually locate the Original Entry Point (OEP), or at least trigger the end of the unpacking process. Depending on the version, UPX uses a loop mechanism to check whether unpacking is complete (Figure 4.2). Once it is, it allocates space for the original program and performs a jump into the "new" unpacked code. We will focus on two versions: first, version 3.91 (2013), which was used to pack the Symbion example we reproduced; and second, version 4.22, the more recent version installed on the machine used for this work. Unfortunately, although it is possible to unpack and repack the Symbion example using UPX version 3.91, it is difficult to pack a recently written and compiled binary with this version. The difference in metadata (such as headers) causes compatibility issues,

often resulting in a segmentation fault[1]. Let's start by opening two files packed with UPX in IDA Pro: the 'elf_packed' file, which uses version 3.91 (on the left in Figure 4.2), and another one, 'sample01', packed with version 4.22 (on the right in Figure 4.2).

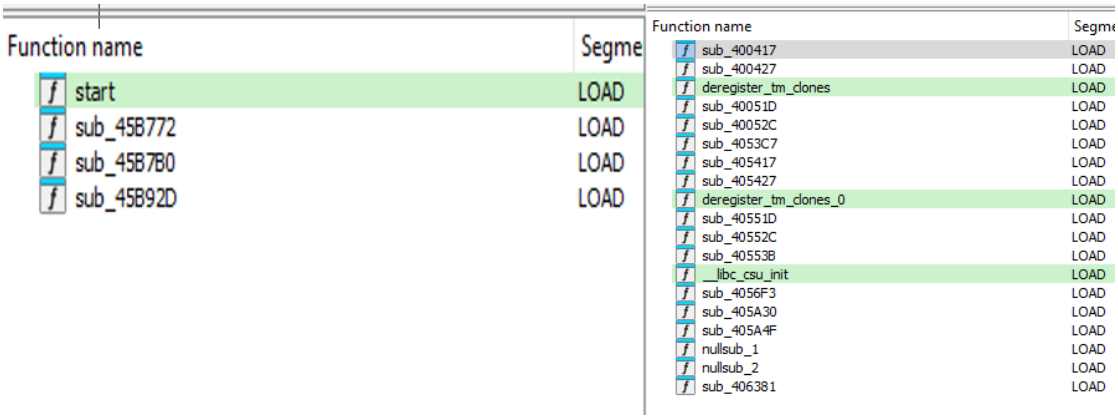

Figure 4.2: Overview of analyses in IDA Pro

We can immediately see that our two files have completely different structures, which can be explained by the difference in UPX versions used. The older 2013 version follows a simpler logic and then is more predictably, the more recent version is significantly more complex. Between UPX versions 3.91 and 4.22, several changes have had a direct impact on analysis and unpacking. First, the decompression stub has been modified: while it was simpler and more readable in 3.91, it became more complex in 4.22, making it harder to identify the real entry point (OEP). Next, relocation handling has been strengthened, which improves compatibility but complicates manual code extraction. Finally, UPX 4.22 expands support to more formats and platforms, requiring unpacking techniques to adapt to more diverse binary structures. We will now attempt to identify different ways to detect the end of the stub. To better understand the behavior of the UPX decompression stub, we can summarize its operation with the following points:

- Allocates memory
- Decompresses the original code into this memory
- Then jumps to the real entry point of the decompressed program

This behavior can indeed be observed in IDA Pro's IDA View-A window. Notably through system calls such as `sys_mmap`, which, following the unpacker's logic,

[1] `https://github.com/upx/upx/issues/106`

allocate memory for the decoded code, particularly a memory range near `0x800000`, which corresponds to the address we aim to detect to trigger the end of the unpacking process (see relevant section).

4.2.1 Finding Addresses

One of the ideas is to detect a jump into the code unpacked. To detect jumps in our packed code, we can start by examining them in IDA Pro, for our file that uses version 3.91 (Figure 4.3). In this figure, we can see on the right the 'start' function, as indicated in Figure 4.2, which makes a call to the function `sub_45B7b0` (on the left in Figure 4.2). This function clearly performs a jump, which could potentially be a transition to the original code. We will use this information later.

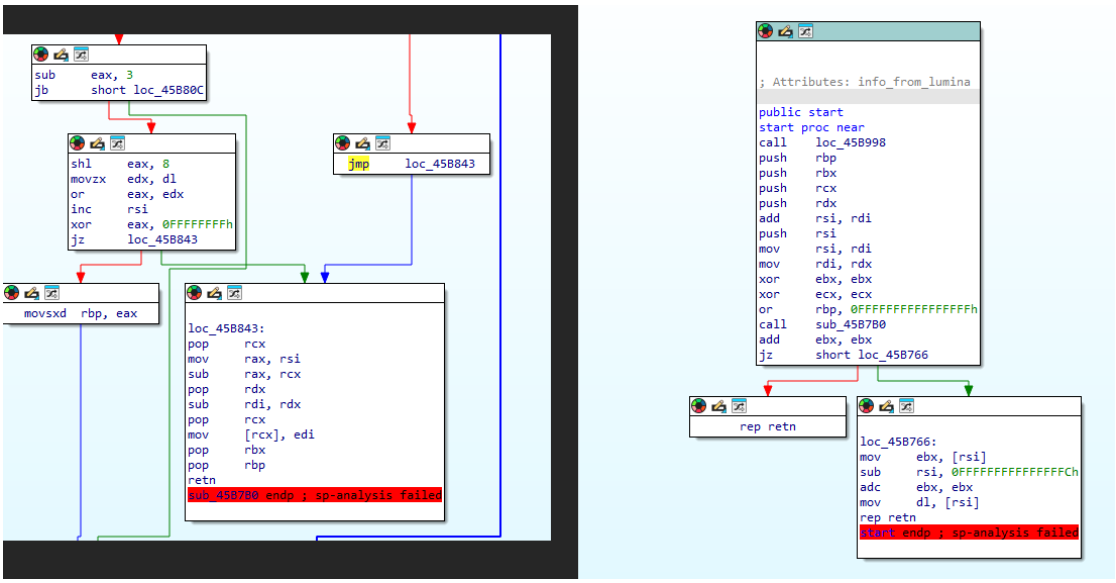

Figure 4.3: Focus on jump instruction in IDA Pro (Symbion example)

If we now observe the binary in GDB using the commands `starti` and `si 1`, we can see that a few distinct addresses are displayed at first, followed by a long sequence of repeated addresses. Let's examine how long this repetition lasts: after 39 instructions, address `0x45b97c` appear, and if we continue stepping through 46,940 instructions, the address changes again for a few more steps before finally reaching addresses like `0x85b745`. Using a small GDB script (shown appendix), we can confirm this behavior. By using script to execute 1000 instructions, we can record the output to a text file and extract the jump instructions using a simple `grep`. Below is a snippet of the script used:

```
gdb -q -x auto.gdb ./packed_elf64 > output.txt
cat output.txt | grep jmp

----Ouput----
=> 0x85b841:   jmp     0x85b7c6
=> 0x85b841:   jmp     0x85b7c6
=> 0x85b841:   jmp     0x85b7c6
=> 0x85b841:   jmp     0x85b7c6
=> 0x85b841:   jmp     0x85b7c6
=> 0x85b841:   jmp     0x85b7c6
```

Now, it is time to analyze our binary packed with UPX 4.22. This version is much more recent and complex, making it more difficult to analyze. Regarding the jumps visible in IDA Pro, we can observe them as shown in Figure 4.4.

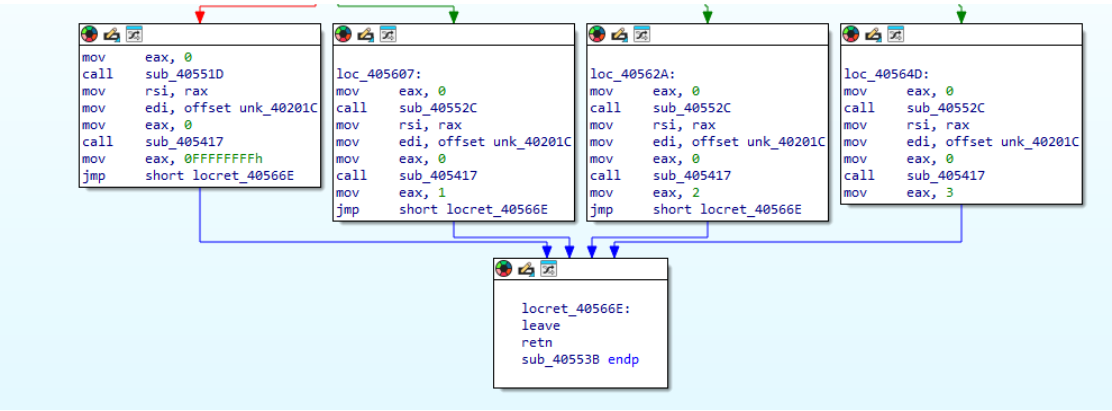

Figure 4.4: Focus on jump instruction in IDA Pro (sample 01 packed)

In IDA Pro View, we can see another jump at address `0x405A2E` (`jmp rax`). Unfortunately, with this address, our code from sample 01 in SEMA reaches a concrete end and fails to find the target. Let's then try using the jump address shown in Figure 4.4. This time, it's good news, it successfully triggers the target address and no longer crashes.

Finally, let's also take a look at what happens in GDB. We can observe that after a series of instructions, a sequence of repeated addresses appears, before the execution eventually returns to different addresses. By using the same script as before, we obtain this output :

```
gdb -q -x auto.gdb ./sample01_packed > output.txt
cat output.txt | grep jmp

----Ouput----
=> 0x4058ba:  jmp    0x4058c4
=> 0x40593f:  jmp    0x4058c4
=> 0x40593f:  jmp    0x4058c4
=> 0x40593f:  jmp    0x4058c4
=> 0x40593f:  jmp    0x4058c4
```

4.3 Dynamic Analysis Approaches

In this section, we aim to explore a more dynamic alternative to manual inspection or GDB scripting. After our static analysis, several possibilities are available to us, such as identifying a jump address, the end of a large loop, or specific syscalls. In our work, and based on the two analyzed examples, we will focus on a different approach. We will attempt to detect a significant difference (jump) in the address space during the unpacking process. Once such a "large" jump occurs within the program's address space, it should indicate that the unpacked code has been loaded, allowing us to trigger the addresses of the original program.

To do this, a small Python script integrable into SEMA is used to initialize a GDB server in the same way as with Symbion. Instructions are then executed one by one for a certain number of cycles (see section 4.2.1). Once the script is finished, the resulting address change can be observed. This code can then be used to update the SEMA configuration file with the detected address, as an argument to trigger the target (see section 3.1.4).

```
python3 uxp_jump.py
---- Output (sample 01) ----
Significant change detected: 0x405838 : 0x405a34
Significant change detected: 0x405a88 : 0x405843
Significant change detected: 0x405944 : 0x405a93
Significant change detected: 0x405a93 : 0x76511c037050
Significant change detected: 0x76511c037113 : 0x76511c03777e
Significant change detected: 0x76511c0377a2 : 0x76511c0371f4
Significant change detected: 0x76511c03727b : 0x40584d

---- Output (packed elf64) ----
Significant change detected: 0x45b740 : 0x45b92e
Significant change detected: 0x45b98c : 0x85b74d
```

4.4 Results

The methodology used to analyze our files packed with UPX is straightforward. First, we need to confirm a few things. The first step is to ensure that the file we want to analyze is indeed packed with UPX. Next, we need to determine the UPX version used, in order to better understand its behavior and facilitate analysis in a reverse engineering tool such as IDA Pro.

Once these steps are completed, we must obtain two key pieces of information before initiating the analysis in SEMA. Since our code is packed, addresses will be overwritten during UPX's unpacking phase. Therefore, we must trigger a specific address at the end of this phase to have a new stub available and to prevent our GDB breakpoint (OEP) from being overwritten.

First, we can use our static analysis and a dynamic Python script to select the addresses. Using `sample01` as our packed (UPX 4.22) file example, we can summarize the results in the following table :

Address	**Works in SEMA**	**Method**
0x405A2E	No	IDA
0x40566E	No	IDA (jmp Figure 4.4)
0x405919	Yes	IDA (jmp after sys memfd create)
0x4058C4	Yes	GDB script
0x40593F	Yes	GDB script
0x405A93	No	Python script
0x4011B9	No	OEP

Table 4.1: Effectiveness of different target addresses with SEMA

By using the addresses that work inside SEMA and the OEP found in section

3.1.3, we successfully analyzed the file in SEMA using Symbion and our earlier implementation. This allowed us to confirm that SEMA correctly identifies all the paths in the file.

Address	**Name**	**GDB address code unpack**
printf	0x40129d	0x40129d <+297>: call 0x401050 <printf@plt> (safe)
printf	0x40127a	0x40127a <+262>: call 0x401050 <printf@plt> (safe)
printf	0x401257	0x401257 <+227>: call 0x401050 <printf@plt> (safe)
printf	0x401234	0x401234 <+192>: call 0x401050 <printf@plt> (evil)

Table 4.2: Output in final_SCDG.json

In conclusion, the combination of static and dynamic analysis allowed us to identify entry points compatible with SEMA for concolic execution. By selecting addresses after the UPX unpacking phase and by carefully placing breakpoints, we ensured that Symbion could operate effectively within SEMA. The results confirmed that SEMA is able to identify all feasible execution paths in the analyzed binary.

Chapter 5

Discussions

5.1 Comparison between symbolic and concolic execution

Symbolic execution alone is a powerful technique that allows for precise control over the execution environment by selecting and delimiting which parts of the program are treated symbolically from the moment the binary is loaded. One of the major advantages of symbolic execution, within SEMA, is its portability across different examples and its ease of adaptation, thanks to its modular structure and ease of integration for unsupported methods. However, a significant drawback of symbolic execution is its susceptibility to evasion techniques. In certain cases (due to path constraints, loop complexity, or specific evasion mechanisms) it can become extremely slow or even impractical to use. This is particularly true in scenarios involving branch explosion, packed binaries, or deliberately obfuscated code.

Concolic execution, on the other hand, makes it possible to reach specific areas of a program that one wishes to analyze. Although this master's thesis focuses on transitioning from concrete to symbolic execution, other approaches, such as exploring new states concretely (through fuzzing) and then analyzing them symbolically, can, in certain cases, offer significant performance improvements and lead to the discovery of new execution paths [22].

Finally, let us review the examples that were tested. After discussing the implementation of concolic execution in SEMA, the packing mechanisms, and the techniques used to identify the critical addresses required for execution, it is now appropriate to evaluate the actual use of this approach. The following paragraph presents the examples we were able to test, while section 5.2 will focus on the limitations encountered.

5.1.1 Results

In this chapter, we present various results to support the identified limitations and the conclusions. The results are analyzed from multiple perspectives: execution time, the quality of the produced analysis, and the generality of the code. To begin with, as shown in Figure 5.1, our implementation using Symbion (concolic execution) is significantly faster than the original version of SEMA (the version used in this work). This can easily be explained by the fact that the unpacking phase is executed concretely.

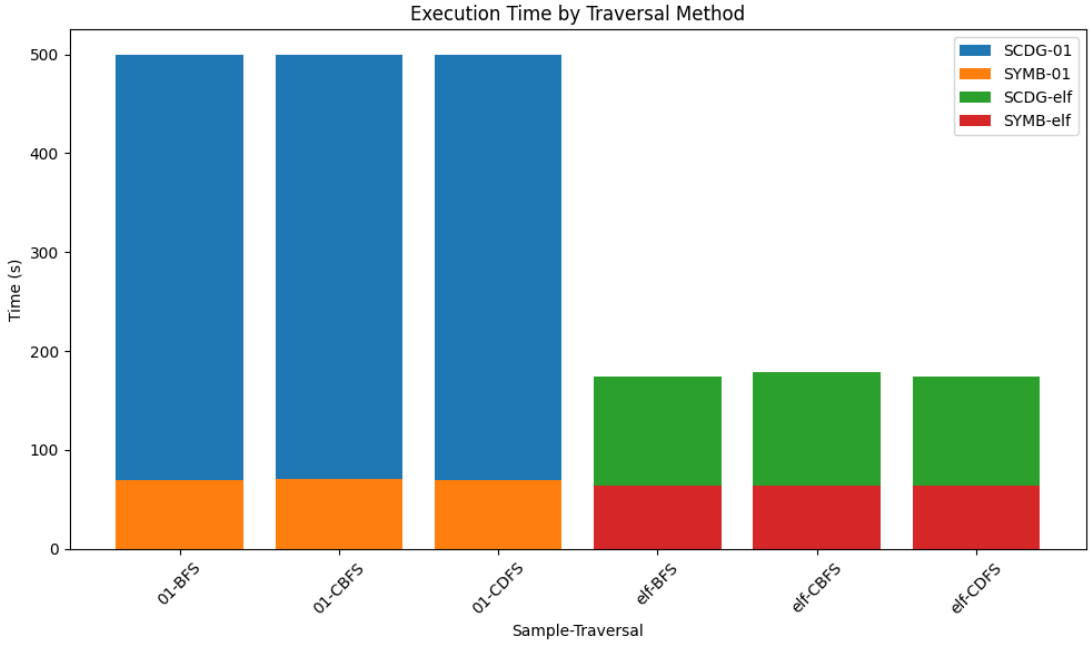

Figure 5.1: Execution time

Figure 5.1 shows the execution time of SEMA with Symbion (SYMB) compared to the initial version without concolic execution (SCDG). The execution time is then analyzed on the Symbion reference example (Section 3.1) and the sample (Section 3.2). We can clearly observe gain of performance. The timings were directly took from SEMA's logs, compiled into a CSV file, and plotted using a small Python script (generated by ChatGPT).

One relevant point is that, during execution without Symbion on our packed examples, the tool ran indefinitely. Although coherent analysis outputs were eventually produced for binary 01, execution was forcibly stopped due to a timeout (parameters in .ini file). This behavior may be caused either by a loss of information (such as the final address) during the packing process, or by a compatibility issue

between SEMA and the version of angr used. For visibility in Figure 5.1, the execution time was set manually to 500 seconds.

Regarding the analysis itself, our approach using Symbion produced the expected output. Refer to Sections 3.1.3 and 3.1.4 for details. We can thus conclude that, for our test cases, the concolic enhanced version of SEMA to reveals different execution paths of the analyzed programs.

Finally, it is important to note that our concolic code is not easily generalizable to other malware samples, due to hardcoded addresses and manually implemented hooking behavior. As it stands, this approach cannot yet be considered broadly applicable.

5.2 Limitations

Several limitations emerged during the course of this work. Many of them represent major obstacles to achieving fully automated execution, while others can be considered minor constraints.

The first limitation stems from the fact that Symbion is no longer actively maintained or at least has lost priority within the angr ecosystem[1]. This has led to the appearance of several bugs and the gradual degradation of certain components. While Symbion previously supported Windows binaries, it has been affected for some time by issues related to the concrete target implementation on Windows systems.

Unfortunately, these limitations are not directly caused by this work, but it is still important to be aware of them. On a more positive note, Symbion remains an open-source project, and contributions from the community may continue to extend and improve this component in the future.

One of the most significant limitation of this work is is the need to hardcode specific addresses in order to execute the analysis correctly. Both the end address of the unpacking phase and the concrete target address that Symbion should reach must be manually defined in the ‘.ini‘ configuration file before launching the analysis. The same challenge applies to the memory region that must be made symbolic. In this work, we explicitly targeted a known address (`rbp-0x4`), but in real-world cases, accessing this kind of information is significantly more complex. Identifying the appropriate memory location to make symbolic is crucial, yet far more difficult to determine when dealing with obfuscated or heavily packed binaries.

This manual setup introduces a strong dependency on reverse engineering tools and techniques, such as debuggers and disassemblers. Such a requirement poses a serious barrier to automation. In practice, most real-world malware samples are

[1] `https://github.com/angr/angr/issues/2701`

significantly more complex than the simplified examples used in this master thesis, and identifying the correct execution points is far from trivial. This limitation greatly reduces the scalability and general applicability of the current approach.

Finally, some information such as symbols, system calls, or hooks, may be lost during the packing process or during the synchronization between concrete and symbolic state. This can lead to incomplete or imprecise symbolic analysis, affecting the quality of the results. While the proposed approach introduces notable improvements, several challenges remain before a reliable and automated solution can be achieved. Nevertheless, this work provides a solid starting point for future improvements to the SEMA framework.

Chapter 6

Conclusion and Future Work

The main objective of this thesis was to enhance the SEMA tool (Symbolic Execution for Malware Analysis) to support concolic execution. In parallel, the goal was to build the foundation for integrating and improving concolic execution within SEMA. This work focused on the reference example presented in section 3.1 and custom samples in section 3.2. This work defined a practical methodology for applying concolic analysis within SEMA, based on initial static and dynamic analysis to find the required information used in the new parameters of the configuration file in sema. Some code was added to the tool (section 3.3), allowing the user to choose whether to analyze the file using Symbion. After testing both the Symbion example and our custom samples with concolic execution, we obtained good results for most files. Despite several technical constraints, we were able to validate certain examples and gain a better understanding of the limitations of Symbion. While many restrictions must be taken into account, we were still able to identify key advantages, including improvements in execution time and the ability to reach execution paths that were previously inaccessible (packed file). Finally, it should be noted that the solution is not optimal. Despite encouraging results, it relies heavily on specific cases and has revealed major issues, such as lost information during synchronization. Therefore, a deeper understanding of these unexpected behaviors is necessary in order to further extend and generalize the solution. It also proposed a straightforward approach for designing custom samples to evaluate the method (section 3.2).

There remains a wide range of future work. Below is a set of directions that were not addressed in this master thesis but would merit deeper investigation. First, a more in-depth analysis of the behavior of UPX and other packers could provide valuable insights into how packing mechanisms affect symbolic and concolic execution. Second, research focused on identifying key addresses, such as the end of the unpacking phase or the concrete target address (EOP), would be highly beneficial. These addresses are currently hardcoded, but developing heuristics or

guided techniques to detect them automatically would significantly enhance the automation and usability of the approach (such as `rbp-0x4`). Lastly, a deeper comparative study between symbolic and concolic execution could help determine how to achieve similar levels of analysis quality in both modes, and under what conditions one should be preferred over the other.

Bibliography

[1] The angr Project. Symbolic expressions and constraint solving. `https://docs.angr.io/en/latest/core-concepts/solver.html`, 2025. Accessed: 2025-05-07.

[2] The angr Project Contributors. Hooks and simprocedures, 2025. Accessed: 2025-05-06.

[3] angr Team. Throwing a tantrum: Part 1 - the angr lifecycle, 2020. Accessed: 2025-05-31.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 633–651. IEEE, 2017.

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

[7] Fabrizio Biondi, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Tutorial: An overview of malware detection and evasion techniques. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, pages 565–586, Cham, 2018. Springer International Publishing.

[8] Cristian Cadar and Martin Nowack. Klee symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, 23:867–870, 2021.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–49, 2012.

[10] SEMA Contributors. Sema - symbolic execution-based malware analysis. `https://github.com/csvl/SEMA`, 2021. Accessed: 2025-05-03.

[11] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, 2016.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3 theorem prover. `https://github.com/Z3Prover/z3`, 2008. Version 4.14.1, accessed May 7, 2025.

[13] Dhruwajita Devi and Sukumar Nandi. Detection of packed malware. In *Proceedings of the First International Conference on Security of Internet of Things*, pages 22–26, 2012.

[14] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008).

[15] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–10, 2020.

[16] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[17] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[18] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64, 2010.

[19] Ravie Lakshmanan. Golden chickens deploy terrastealerv2 to steal browser credentials and crypto wallet data. *The Hacker News*, May 2025.

[20] Serena Lucca. Let's analyze malware with symbolic execution: a practical study. Master's thesis, Ecole polytechnique de Louvain, 2022. Master [120] : ingénieur civil en informatique, à finalité spécialisée.

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

[22] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, 2007.

[23] Benjamin Marais, Tony Quertier, and Christophe Chesneau. Malware analysis with artificial intelligence and a particular attention on results interpretability. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 43–55. Springer, 2021.

[24] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019.

[25] Markus Oberhumer, László Molnár, and John Reiser. Upx - the ultimate packer for executables, 2025. Accessed May 2025.

[26] V Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar. Signature generation and detection of malware families. In *Information Security and Privacy: 13th Australasian Conference, ACISP 2008, Wollongong, Australia, July 7-9, 2008. Proceedings 13*, pages 336–349. Springer, 2008.

[27] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications*, SSTIC, pages 31–54, Rennes, France, June 2015.

[28] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13*, pages 270–284. Springer, 2011.

[29] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[31] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[32] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[33] Angr Team. Introducing angr-symbion: A symbolic execution framework for python. `https://angr.io/blog/angr_symbion/`, 2021. Accessed: 2025-05-06.

[34] Alexey Vishnyakov, Daniil Kuts, Vlada Logunova, Darya Parygina, Eli Kobrin, Georgy Savidov, and Andrey Fedotov. Sydr-fuzz: Continuous hybrid fuzzing and dynamic analysis for security development lifecycle. In *2022 Ivannikov ISPRAS Open Conference (ISPRAS)*, pages 111–123. IEEE, 2022.

[35] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing packed malware. *IEEE Security Privacy*, 6(5):65–69, 2008.

[36] Eric Youngdale. Kernel korner: The elf object file format by dissection. *Linux Journal*, 1995(13es):15–es, 1995.

[37] Li Zhang and Vrizlynn L. L. Thing. A hybrid symbolic execution assisted fuzzing method. In *TENCON 2017 - 2017 IEEE Region 10 Conference*, pages 822–825, 2017.

Appendix A

Github

All the code mentioned in this work can be found at:`https://github.com/tbouvencourt/TFE_Enhancing_Malware_Analysis_with_Concolic_Execution`

Appendix B

Script gdb

```
starti

define step
  si 46900
  set $i = 0
  while $i < 100
    x/i $rip
    si 1
    set $i++
  end
end

step
quit
```

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