UCLouvain

École polytechnique de Louvain

Practical Ransomware Analysis with symbolic execution

Author: **Richard DE JAMBLINNE DE MEUX**
Supervisor: **Charles PECHEUR**
Readers: **Charles-Henry BERTRAND VAN OUYTSEL, Kim MENS**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

This thesis explores the application of symbolic execution to malware analysis, focusing on the `WannaCry` and `GonnaCry` ransomware families. Using the `SEMA` toolchain in combination with `Ghidra`, we analyzed these ransomware samples both statically and symbolically to investigate their behavior and identify external function calls.

While we successfully achieved full path coverage for `GonnaCry` and one of the three components of `WannaCry`, the analysis also revealed several limitations and challenges associated with symbolic execution. To support this effort, we implemented nearly one hundred custom `SimProcedures`, developed a custom plugin, and integrated an additional plugin from a different version of the toolchain. Furthermore, we proposed several improvements aimed at enhancing symbolic execution capabilities and addressing the identified shortcomings.

Our findings highlight both the potential and the limitations of symbolic analysis in the context of ransomware research, offering insights into its effectiveness and areas for future development.

Acknowledgments

Disclaimer

Artificial intelligence tools (like ChatGPT), were used during the preparation of this thesis. These tools primarily assisted in proofreading tasks, such as correcting spelling, grammar, and improving the clarity and coherence of the text. Additionally, AI was employed to implement certain code functionalities, clarify the operation of specific code segments, and assist in diagnosing errors linked to the tools and software libraries used during the development process.

Contents

1 Introduction **5**

2 Background and Related work **7**
2.1 Malware Analysis Techniques 7
2.1.1 Static Analysis 7
2.1.2 Dynamic Analysis 8
2.1.3 Symbolic Execution 8
2.1.4 Concolic Execution 10
2.2 Analysis Tools 10
2.2.1 Ghidra 10
2.2.2 RetDec 11
2.2.3 Angr 12
2.3 Ransomware 13
2.3.1 Definition and Classification 14
2.3.2 History and Evolution 14
2.3.3 General Workflow and Propagation 15
2.3.4 Case Study 16
2.4 Sema Toolchain 16
2.4.1 SEMA-SCDG 16
2.5 Related work 18
2.5.1 Ransomware analysis 18
2.5.2 Symbolic Execution 19
2.5.3 Symbolic Execution for Malware Analysis 19
2.5.4 SEMA-Related Work 20
2.6 Summary 20

3 Gonnacry **21**
3.1 Functional Overview 22
3.2 Sema execution 23
3.2.1 Faulty `SimProcedure` 23
3.2.2 Symbolic execution limitation 25
3.2.3 Missing `SimProcedure` 27
3.3 Call Coverage Analysis 29

3.4 Conclusion 31

4 WannaCry **32**
4.1 `mssecsvc.exe` 33
4.1.1 Functional Overview 34
4.1.2 SEMA Execution 34
4.1.3 Result Analysis 36
4.1.4 Worm Insights 37
4.2 `tasksche.exe` 38
4.2.1 Functional Overview 38
4.2.2 SEMA Execution 40
4.2.3 Result Analysis 44
4.3 `kbdlv` DLL 45
4.3.1 Functional Overview 45
4.3.2 SEMA Execution 47
4.3.3 Result Analysis 50
4.4 Other sample 51
4.5 Conclusion 53

5 Conclusion **55**

Chapter 1

Introduction

According to the European Union Agency for Cybersecurity (ENISA), ransomware continued to represent one of the most significant cybersecurity threats in 2024. This observation was highlighted in ENISA's 2024 Threat Landscape Report, which provides a comprehensive overview of emerging cyber threats and trends [1].

In 2025, BlackFog, a company specialized in cybersecurity, reported in their monthly update that in January alone, they recorded 92 disclosed ransomware attacks targeting various organizations around the world [2]. This represents a 21% increase compared to the same period last year. Ransomware, being a type of malicious software (malware) that targets private data to capture and hold it hostage for ransom, poses a serious threat. In its report, BlackFog highlighted that each attack carries a monetary risk ranging from tens of thousands to hundreds of millions—or even billions—of dollars. Data is not safe either, as many attacks involve data theft and resale, with stolen datasets reaching sizes of several terabytes.

Currently, the two primary methods for analyzing malware are static analysis and dynamic analysis [3]. On the one hand, static analysis relies on syntactic signatures found within the source code; on the other, dynamic analysis runs the malware in a sandboxed environment to analyze it. These two approaches have their advantages and disadvantages: static analysis is vulnerable to obfuscation and syntactic patterns can easily be altered; dynamic analysis explores only one execution path, which can be limited or misleading if the malware detects the sandbox environment. These weaknesses make malware analysis increasingly challenging for cybersecurity professionals.

In this work, we explore an alternative technique called symbolic execution. This method avoids the limitations of the previous two approaches by simulating the execution of the malware across all possible paths rather than just one. While symbolic analysis is more complex to implement and is susceptible to the path explosion problem, it can facilitate the detection of malicious behavior that may remain hidden to both dynamic and static analysis techniques.

To implement symbolic execution, we utilize the "Symbolic Execution for Malware Analysis" (SEMA) toolchain, developed by Bertrand Van Ouytsel et al. [4], with a particular focus on its SCDG (System Call Dependency Graph) module. This module is designed to

perform symbolic execution on software, explore all possible execution paths, and collect detailed information about the binary. The collected data primarily consists of information related to external calls encountered by SEMA-SCDG during execution. After the analysis, this data is output as both a System Call Dependency Graph (SCDG) and a JSON file. Afterwards, this data can be used by SEMA-Classifier to recognize and classify malware, all within the broader context of malicious software recognition and classification.

In its current version, the SEMA toolchain is capable of analyzing certain types of malware; however, it lacks dedicated support for ransomware. As a result, when ransomware samples are analyzed, there is a significant likelihood of failure or incomplete analysis. In many cases, these issues manifest in two distinct ways: the tool may crash and produce no output, or it may generate a partial analysis that fails to reveal the malicious behavior. As we will discuss in Chapter 3, some progress has been made toward integrating ransomware analysis into SEMA, but these improvements have not yet been incorporated into the current version.

The goal of this work is to expand the capabilities of the `SEMA` toolchain to analyze two ransomware families targeting both Linux and Windows platforms. More specifically, our objective is to understand how these ransomware samples operate and to ensure that the `SEMA-SCDG` module explores all possible execution paths for each family—or, in cases where full path coverage is not achieved, to identify the reasons for this limitation.

To validate the symbolic execution results, we compare the output produced by the `SCDG` module with the external calls and binary addresses identified through static analysis.

To achieve this, the thesis is structured into five parts. Following this introduction, Chapter 2 provides the necessary background knowledge to support the research, including an overview of ransomware, the tools used, and the `SEMA` toolchain. This chapter also presents related work in the field.

Chapters 3 and 4 focus on the two ransomware families analyzed in this study. Each chapter details the analysis process using `SEMA`, the efforts made to achieve complete path coverage, and the challenges encountered.

Finally, Chapter 5 summarizes the results, discusses potential improvements to the toolchain, and concludes the study.

Chapter 2

Background and Related work

In this chapter, we present the theoretical knowledge that will be useful in the following chapters. We begin by exploring various techniques used to analyze malware, with a particular focus on symbolic analysis, which constitutes the core method applied throughout this work. Subsequently, we introduce the main external tools and libraries used during the implementation. We then provide an overview of ransomware, followed by a detailed presentation of the `SEMA` toolchain. Finally, we conclude the chapter by discussing studies and related work relevant to this thesis.

2.1 Malware Analysis Techniques

In this section, we discuss four different techniques, starting with the simplest—static and dynamic analysis [5, 6]. We then move to the main focus of this work: symbolic execution [7], followed by a more advanced technique, concolic execution [8, 9].

2.1.1 Static Analysis

Static analysis refers to the process of analyzing a program without executing it. This can be performed in various ways, but one of the most common methods involves disassembling the program. Disassembly transforms the raw binary into assembly code, which is a low-level representation of the program's instructions.

This assembly code can then be analyzed to identify syntactic signatures—recognizable patterns in the code such as checksums, specific strings, API calls, and other features. These signatures can be used to create detection rules for tools like `YARA` (Yet Another Ridiculous Acronym) [1], `DIE` (Detect It Easy)[2], or `PEiD` (Portable Executable iDentifier) [3]. These rules encapsulate the identified patterns and enable lightweight classification of malware families through pattern matching.

[1] https://virustotal.github.io/yara/

[2] https://github.com/horsicq/Detect-It-Easy

[3] https://github.com/wolfram77web/app-peid

However, this approach to recognizing malicious code has a major limitation: syntactic signatures, while easy to detect, are also easy to bypass with small modifications or obfuscation. A small change in a string or code pattern can invalidate a rule, potentially leading to misclassification. Despite this drawback, YARA rules remain popular due to their ease of use and fast performance.

Beyond simple signature detection, static analysis can also be used to understand the internal logic and control flow of a program. While analyzing disassembled code directly is possible, raw assembly is often difficult to read and interpret. Tools like Ghidra and RetDec not only disassemble binaries but also decompile them into higher-level representations, making them more accessible for human analysis.

In our analysis, static examination using Ghidra's decompilation capabilities provides valuable insights into the malware family under investigation. It not only enhances our overall understanding of the sample's structure and behavior but also helps identify specific code segments and addresses that lead to issues during symbolic execution.

2.1.2 Dynamic Analysis

Dynamic analysis involves analyzing a program by executing it. Since running potentially malicious software on a host system can be dangerous, this analysis is typically performed within a controlled environment—commonly a virtual machine (VM) acting as a sandbox.

Executing the program inside a VM allows analysts to observe the binary's behavior at various levels, such as monitoring memory usage, file system changes, or network activity. These observations can be used to label the binary or assign it a score indicating the likelihood that it is malicious.

However, dynamic analysis typically explores only a single execution path through the program. As will be discussed in Chapter 4, which focuses on the WannaCry malware family, sophisticated malware developers can implement sandbox-evasion techniques. These techniques allow the malware to detect that it is running inside a virtualized environment and behave in a non-malicious or dormant way, rendering the analysis ineffective.

Additionally, while sandboxes are designed to isolate malicious behavior, they are not flawless. Vulnerabilities in virtualization software can sometimes be exploited by advanced malware to escape the sandbox and infect the host system. Although such cases are rare and typically patched quickly by vendors, they still represent a potential risk during dynamic analysis.

2.1.3 Symbolic Execution

Symbolic execution is a technique used to analyze a binary without actually executing it. Instead, it simulates the program's execution by replacing concrete inputs with symbolic variables. This abstraction allows the analysis to reason about multiple possible inputs simultaneously and explore various execution paths through the binary.

Unlike dynamic analysis, which only follows a single execution path determined by specific input values, symbolic execution can theoretically explore the entire control flow of

a program. This makes it particularly effective at bypassing sandbox evasion techniques implemented by malware developers.

To achieve this, symbolic execution relies not only on concrete variables—i.e., variables that hold a single, well-defined value—but also on symbolic variables. Symbolic variables do not contain a specific value; instead, they represent a set of constraints that define the range of possible values the variable can take.

For example, consider a symbolic variable `x`. When the program encounters an `if` statement involving `x`, symbolic execution generates two separate execution paths: one where `x` satisfies the condition and the program enters the conditional block, and another where the condition is not satisfied and the block is skipped. In each case, the symbolic execution engine updates the constraints on `x` to ensure consistency with the branching condition.

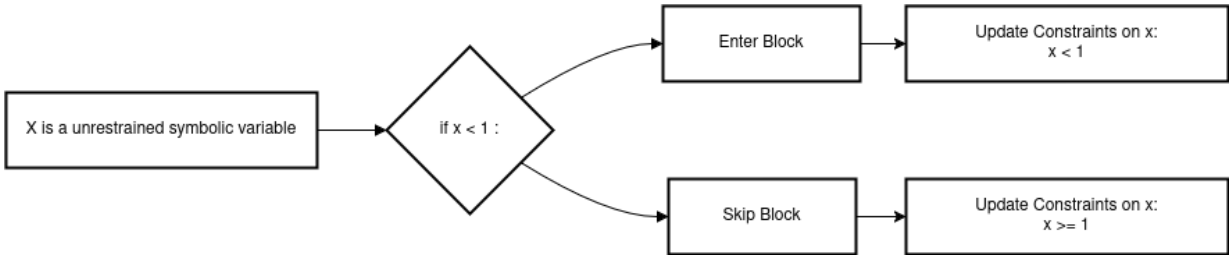

Figure 2.1: Illustration of branching behavior.

However, symbolic execution has significant drawbacks:

On the one hand, because symbolic execution is an abstract form of execution, all system calls, library functions, and interactions with the operating system must be simulated. These external calls must be either modeled or emulated; otherwise, they are generally skipped and replaced with an unconstrained symbolic variable. As a result, the internal behavior of these calls is not reproduced, which can lead to inaccuracies or errors during execution. Moreover, the use of unconstrained symbolic variables can significantly slow down the execution process due to the added complexity in constraint solving.

On the other hand, symbolic execution is particularly vulnerable to the well-known problem of *path explosion*. Each conditional statement—such as an `if` condition or a loop—can generate new execution branches, resulting in exponential growth in the number of paths that must be explored. This rapid increase in execution paths can significantly slow down the analysis process, or even cause it to crash due to memory exhaustion or time constraints. In unfavorable scenarios, even a simple `for` loop containing multiple `if` statements can quickly lead to dozens of distinct execution paths, severely impacting performance and scalability.

Despite these challenges, symbolic execution can yield more precise insights for specific samples than static or dynamic analysis alone, even though the latter techniques are generally more scalable. In this work, symbolic analysis is a central focus and will be

conducted using the angr library, integrated within the SEMA toolchain, which will be described in more detail later in this chapter.

2.1.4 Concolic Execution

Concolic execution is a hybrid approach that combines concrete and symbolic execution—hence the name "concolic." As the name implies, this technique blends the strengths of both methods. It helps to mitigate the risk of path explosion while allowing better control over path exploration, enabling the analysis of multiple execution paths.

There are several ways to implement concolic execution, but the core idea is to execute parts or the entirety of the code with *both* concrete values and symbolic expressions. Information gathered from the concrete execution can guide the symbolic analysis, and vice versa, allowing for more efficient and targeted exploration.

Multiple research efforts and tools have been developed based on this approach. Notably, CUTE [9], a Concolic Unit Testing Engine, employs concolic execution to automatically generate unit tests for C programs. In contrast, Symbion [8], an extension of angr, enables dynamic switching between concrete and symbolic execution. This capability is particularly valuable in malware analysis, as it allows sections of code that are too resource-intensive or complex for symbolic execution to be executed concretely.

2.2 Analysis Tools

In this section, we present three tool frameworks used throughout Chapters 3 and 4. First, we introduce Ghidra and RetDec, two tools commonly used for static code analysis. Then, we discuss angr, a powerful framework designed to perform symbolic execution.

2.2.1 Ghidra

Ghidra is a software reverse engineering framework developed by the National Security Agency (NSA), with its initial public release in 2019. The framework is available on its official website [4] and its GitHub repository [5].

This tool is widely used to disassemble binaries into assembly code and to decompile them into C code, which is generally easier to read and analyze. Figure 2.2 illustrates the analysis of a sample from the Gonnacry ransomware (which will be discussed later in Section 3). The left pane displays the disassembled assembly code, while the right pane shows the corresponding code decompiled into C.

In this example, we can see the original function names because the binary was not stripped of its symbols, allowing Ghidra to recover these names. In real-world scenarios, however, binaries are often stripped to reduce file size and increase obfuscation. This results in generic function names such as FUN_00420000. On the other hand, system calls and

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

[5] https://github.com/NationalSecurityAgency/ghidra

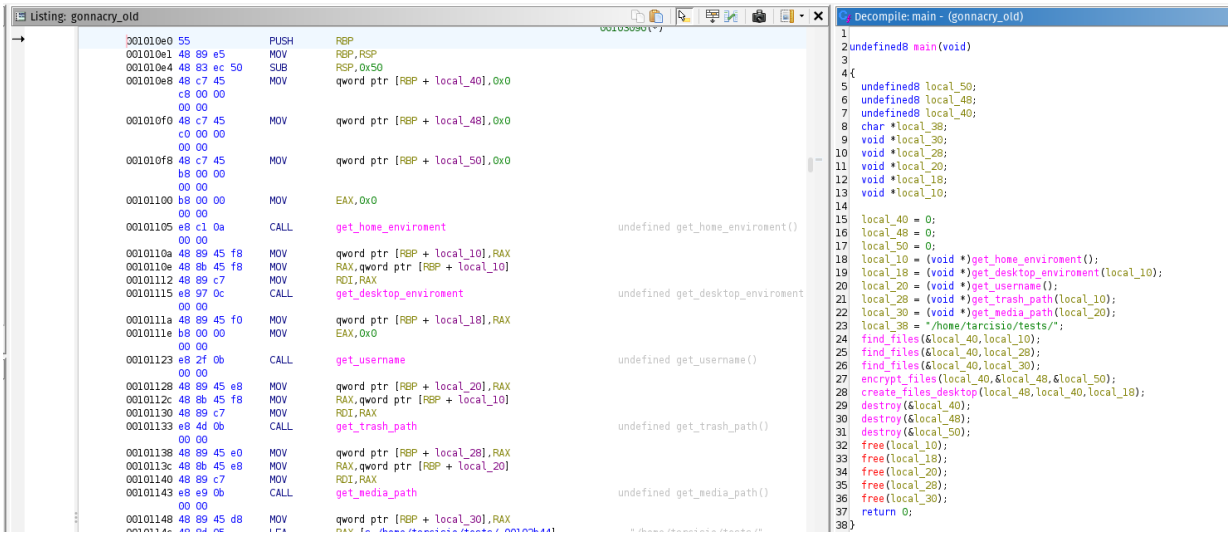

Figure 2.2: Ghidra disassembly and decompilation view

standard API calls—such as the `free` function shown in the example—are usually still recognizable, even in stripped binaries. This is because Ghidra can identify standard library functions based on known patterns or signatures.

These two views were the most frequently used during this work. The disassembled view (left) is particularly useful for examining specific addresses that may lead to errors during symbolic execution. By correlating these problematic addresses with the decompiled view (right), we can better understand the program's work flow and the source of the error. Additionally, the decompiled code provides a high-level overview of the binary's overall behavior, making it easier to understand how the malware operates.

2.2.2 RetDec

`RetDec` is an open-source decompiler and disassembler originally developed by AVG and later acquired by Avast, which currently maintains the project. However, as of now, it is under "limited maintenance mode." The tool is available on its GitHub repository [6].

Although similar in purpose to `Ghidra`, `RetDec` operates in a slightly different manner. While `Ghidra` provides an interactive graphical interface for reverse engineering, `RetDec` generates a set of output files as the result of its analysis. These files include the disassembled code, the decompiled C-like code, and various reports from static analysis.

Although its capabilities are less advanced compared to `Ghidra`, `RetDec`'s ability to quickly and automatically generate various informational files makes it a valuable tool for rapidly extracting information. In this work, the generated files are used in conjunction with small Python scripts to extract external function calls along with their corresponding

[6] `https://github.com/avast/retdec`

instruction addresses. This information is then compared with the output of `SEMA` to quickly assess the current state of symbolic execution. Further details will be provided in Chapter 3.

2.2.3 Angr

`Angr` [10] is an open-source binary analysis framework, mainly used for symbolic execution. It is written in Python and can be found on its GitHub page[7] or on its official website [8].

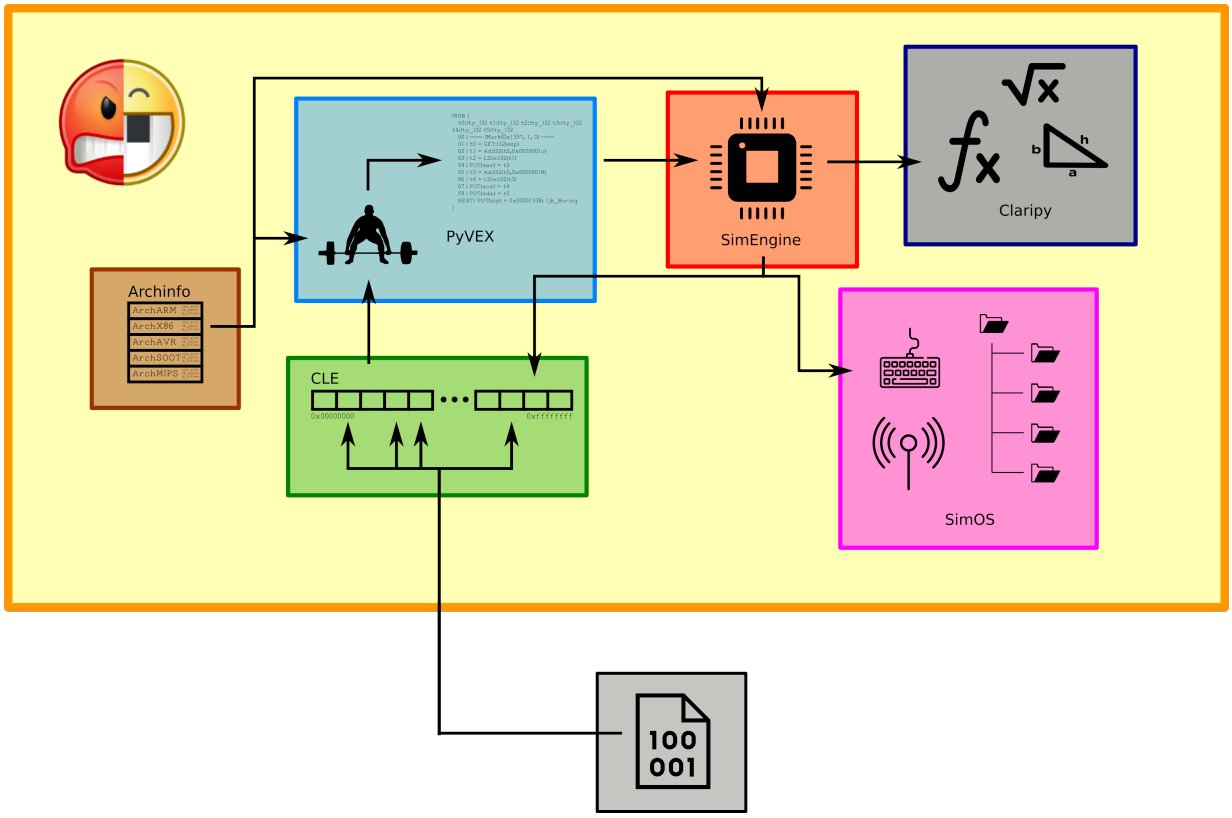

Figure 2.3: angr lifecycle [11]

As shown in Figure 2.3, found in one of the articles from the angr blog [11], `angr` is composed of six main components that work together:

In the first step, **CLE (CLE Loads Everything)** is called to load and analyze the binary. It is responsible for detecting the architecture and organizing the memory layout required for further analysis by angr.

The architecture information is then passed to **Archinfo**, which retrieves an `Arch` object containing details such as register mappings, bit width, endianness (i.e., how bytes are

[7] `https://github.com/angr/angr`
[8] `https://angr.io/`

ordered in memory), and other architecture-specific attributes.

To support multiple architectures, `angr` does not execute machine code directly. Instead, it uses an intermediate representation called `VEX`. The base code is transformed into the `VEX` representation through a process known as *lifting*, which is handled by **PyVEX**.

After lifting, the **SimEngine** interprets the VEX instructions. It takes a program state and applies a block of VEX instructions to it. This results in the creation of multiple successor states, each representing a possible outcome of the execution. When a branch is encountered, new states are generated by adding constraints that represent the conditions required for each path, allowing all potential paths to be explored.

As explained earlier, symbolic variables are a crucial component of symbolic execution. These variables form the cornerstone of the technique and can quickly become highly complex, as each one may carry a set of intricate constraints. For this reason, `angr` incorporates its own solver, called **Claripy**.

Claripy's role is not limited to solving these complex constraint sets; it is also responsible for creating and composing them. As illustrated in Figure 2.4, Claripy is used both as a solver (line 1) and to create the symbolic variable (line 2). When we add the constraint $a > 5$ (line 3) and the constraint $a < 10$ (line 5), and then request the evaluation of variable `a`, the solver returns all possible values that satisfy the constraints.

```
1 >>> s = claripy.Solver()
2 >>> a = claripy.BVS("a",64)
3 >>> s.add(a > 5)
4 (<Bool a_0_8 > 5>,)
5 >>> s.add(a < 10)
6 (<Bool a_0_8 < 10>,)
7 >>> s.eval(a,10)
8 (6, 7, 8, 9)
```

Figure 2.4: Claripy symbolic constraint example

Finally, **SimOS** creates an environment that matches the operating system detected by CLE. It also initializes the first program state, which will later be used by the `SimEngine`. This environment supports the emulation of system calls and standard library functions using what are called `SimProcedures`, which we will discuss further in Section 2.4.

2.3 Ransomware

In this section, we define what ransomware is and present the different types that exist. We then explore its history and evolution over time. Following that, we describe the general workflow of ransomware attacks. Finally, we briefly introduce the specific families that we will analyze in this work.

2.3.1 Definition and Classification

A ransomware (ransom software) is a subset of malware (malicious software) designed to restrict access to a system or data until a requested ransom amount from the attacker is satisfied, as defined by Oz et al. [3]. Currently, ransomware can be classified into three major categories [12, 13]:

Scareware: This is the least harmful type of ransomware. It relies on psychological manipulation to coerce users into installing fake software or purchasing suspicious antivirus tools, which may themselves be malware. The coercion typically takes the form of pop-ups or warning messages claiming that the device is infected. While it does not actually harm the system if ignored, scareware leverages fear rather than technical damage. Initially, it was not classified as ransomware, but due to its widespread use, it is now considered part of the ransomware family.

Lock Screen or Locker: This type of ransomware locks the device itself by preventing access to the user interface, sometimes through a full-screen window, other times by blocking keyboard input. It may display instructions on how to unlock the system, usually involving a ransom payment. Despite this, lockers are generally easy to bypass by restarting the computer in safe mode or using antivirus tools.

Crypto: This is the most dangerous type of ransomware. It encrypts the victim's data using strong encryption algorithms, making it nearly impossible to recover the files without the decryption key. The attacker demands a ransom, often in cryptocurrency to preserve anonymity and avoid traceability. Upon payment, victims may receive a decryption key to recover their data.

In the remainder of this work, we focus primarily on crypto-ransomware due to its technical complexity, impact, and the damage it can cause.

2.3.2 History and Evolution

The concept of ransomware is not new[14]. The first known ransomware appeared in 1989 and was created by Joseph Popp. This malware, named `AIDS`, was a Trojan horse distributed via floppy disks. Once executed, it encrypted files on the `C:` drive and demanded a payment of $189. However, it was not very effective due to several factors:

- The limited number of potential victims and the slow, manual infection method.
- The weak encryption algorithm, which was easily reversible.
- Ineffective payment collection methods.
- At the time, computers were not widely used, and the value of digital data was relatively low.

Over the decades, these limitations have been overcome through technological advancements, contributing to the modern threat landscape. Four key factors have significantly evolved:

Internet: While the internet connects people globally, it also provides a perfect environment for malware propagation. It allows ransomware to spread quickly and infect a wide range of systems with minimal effort.

Encryption Methods: Modern encryption libraries are widely available and easy to implement. This reduces the skill level required to create ransomware and ensures strong, virtually irreversible encryption, making recovery without the decryption key almost impossible.

Cryptocurrency: The rise of cryptocurrencies like Bitcoin has enabled anonymous and untraceable financial transactions. This has made them the preferred payment method for ransomware demands, ensuring the anonymity of the attackers.

Data Value: Today, computers store vast amounts of valuable personal and professional data. This greatly increases the leverage attackers have, as victims are more likely to pay to recover critical files.

All of these evolutions have contributed to a continuous rise in ransomware incidents, increasing not only their frequency but also their severity.

2.3.3 General Workflow and Propagation

In general, the workflow of a ransomware attack can be divided into four main stages [13]:

Installation: At this stage, the ransomware first arrives on the target machine. It typically modifies registry keys, alters files in directories such as `AppData`, and ensures persistence on the system.

Communication: Once installed, the ransomware connects to a command and control (C&C) server controlled by the attacker. This communication is used to generate or retrieve a secure encryption key without leaving any trace of it on the victim's device.

File Search and Encryption: Using the encryption key obtained, the ransomware searches for files on the system and encrypts them, effectively locking the user out of their own data.

Extortion: After encryption is complete, the ransomware displays a ransom message with payment instructions. Some variants allow victims to decrypt a few files as proof, while others include countdown timers that threaten permanent deletion of the decryption key if the ransom is not paid.

2.3.4 Case Study

In this work, we analyze two ransomware families:

Gonnacry: The first ransomware studied, discussed in Chapter 3, is `Gonnacry`, an open-source, Linux-based ransomware.

Wannacry: The second case, covered in Chapter 4, is `Wannacry`, a high-profile Windows ransomware that exploited a vulnerability originally discovered by the NSA.

2.4 Sema Toolchain

Sema is an open-source malware analysis and classification toolchain created by Bertrand Van Ouytsel et al. [4]. This toolchain uses symbolic execution to analyze and classify malware. It is the primary focus of this work, as we aim to use and improve it to enhance ransomware analysis.

Sema is composed of three components: the SCDG, the Classifier, and the Federated Learning module. Since the main focus of this thesis is the SCDG, we will briefly explain the other two components before discussing the SCDG in detail.

SEMA-Classifier This component is responsible for detecting and classifying malware. It takes the output of the SCDG execution as input and uses various classification models to identify malware. These models are based on techniques such as graph mining, graph kernels and support vector machines, or deep learning. SEMA is designed to be modular, allowing for the easy addition of new classifiers.

SEMA-Federated Learning This component enables the creation of a secure network of multiple devices communicating through a central server. Each device trains its own deep neural network on local data, and only the model parameters are shared with the server. The server then securely aggregates the parameters to improve the global model and distributes the updated model back to the devices. This approach preserves data privacy by ensuring that raw data is never shared.

2.4.1 SEMA-SCDG

SEMA-SCDG is the component that uses the `angr` framework as a foundation to enable symbolic execution of various binaries. In its current form, SEMA-SCDG supports `ELF`

and PE binary files but cannot process .NET binaries. After loading the binary, it performs symbolic execution using multiple components to create a System Call Dependency Graph (SCDG) and its corresponding JSON file, which represents interactions between different external calls. As mentioned earlier, these SCDGs are later used by the SEMA-Classifier to detect and classify malware.

The components we refer to are implemented in four different forms: exploration methods, plugins, SimProcedures, and hooks. All components are highly modular and can be easily modified, added, or removed. During this work, we will create or implement new plugins, SimProcedures, and hooks. The existing exploration methods, such as basic and custom DFS and BFS, are sufficient and do not require modifications.

Plugins Like traditional plugins, SEMA-SCDG plugins extend the base tool with additional functionality. They can serve various purposes. For example, plugin_ioc_report generates an Indicators of Compromise (IOC) report after execution. plugin_hooks enables the use of hooks during symbolic execution. plugin_resources initializes data structures used by SimProcedures.

SimProcedures Since SEMA-SCDG relies on symbolic execution, encountering unknown external calls can pose significant challenges. When such a call is encountered, angr may generate an unconstrained symbolic value without simulating the actual behavior of the function. To address this, angr provides *SimProcedures*—modular code blocks that allow users to simulate the behavior of specific external functions.

For example, consider the strlen function, which returns the length of a string. Without a corresponding SimProcedure, angr may treat the return value as an unconstrained symbolic variable. If this value is later used to define the size of an array, it could lead to the creation of an excessively large array, consuming unnecessary memory and potentially distorting the analysis. By contrast, a properly implemented SimProcedure can return the actual length of the string, conserving resources and improving the accuracy of the analysis.

SimProcedures are modular and easily integrated into SEMA by adding a properly named and structured Python file. During execution, the tool automatically substitutes recognized external calls with their corresponding SimProcedures. It is important to note that SEMA uses two distinct sets of SimProcedures, depending on whether the analyzed malware is built for Windows or Linux.

Hooks Hooks function similarly to SimProcedures and can replace parts of the execution with custom code. Unlike SimProcedures, which replace function calls based on their names, hooks use instruction addresses. This makes them useful for replacing specific functions or instructions within the binary. As mentioned earlier, using custom hooks in SEMA requires activating the plugin_hooks.

Configuration SEMA-SCDG requires a configuration file, where users can specify the target binary, the exploration method, and which plugins to activate. SimProcedures and

hooks are used automatically. To disable a SimProcedure, its corresponding file must be removed. Disabling hooks requires both file removal and modifications to the SCDG code.

Output As explained earlier, SEMA-SCDG produces two outputs: the System Call Dependency Graph (SCDG) and a corresponding JSON file containing detailed information. While the SCDG representation is used by the classifier, we will use the JSON file for our analysis, as it contains more useful information and is easier to work with. This file includes all the external calls that SEMA intercepts and simulates. In addition, it contains contextual information about these calls, such as their addresses within the binary and their arguments. It also documents the relationships between calls—for example, if the output of one call is used as the input for another. These relationships are what the SCDG graphically represents.

2.5 Related work

This section presents prior work and research related to our study. It is organized into four main areas: first, general research on ransomware analysis; second, foundational work on symbolic execution; third, the application of symbolic execution to malware analysis; and finally, existing studies specifically focused on the `SEMA` toolchain.

2.5.1 Ransomware analysis

Malware analysis—and more specifically, ransomware analysis—has become extensively studied in recent years, resulting in a significant number of academic publications. Notably, Helen J. Chittooparambil et al. [13] conducted a focused study on the classification and evolution of ransomware, highlighting its operational stages and assessing a selection of detection techniques, such as UNVEIL and honeypot-based systems. Their work emphasizes the critical limitation of existing solutions, particularly their inability to detect ransomware during the early stages of infection, and calls for the development of proactive detection methods.

A more recent and comprehensive contribution is provided by Harun Oz et al. [3], who present an in-depth survey covering over a hundred academic works published between 1989 and 2020. The authors analyze over 40 notable ransomware families across various platforms, including PCs, mobile devices, and IoT/CPS environments. Their taxonomy categorizes ransomware by infection vectors, encryption mechanisms, command and control communication, and extortion methods. Moreover, the study offers an extensive overview of existing defense strategies—including static and dynamic analysis, detection using machine learning, and recovery mechanisms—and identifies key challenges and open research problems such as human-operated attacks, fileless ransomware, and platform-independent threats. Their findings provide a holistic understanding of the evolving nature of ransomware and the current gaps in defense research.

2.5.2 Symbolic Execution

Symbolic execution was first introduced in 1976 by James C. King [15] as a technique for exploring program behavior by treating inputs as symbolic values rather than concrete ones. Since then, it has become the subject of extensive research.

Notable contributions include the work of Cadar and Sen [16], who provide a comprehensive overview of the evolution of symbolic execution. Their work highlights key developments such as concolic execution—a hybrid approach that combines concrete and symbolic execution. They present tools like CUTE (Concolic Unit Testing Engine) [17], which leverages concolic execution to automatically generate unit tests, and KLEE [18], which uses pure symbolic execution to detect bugs in programs. The authors also discuss major challenges currently facing symbolic execution, including path explosion and constraint solving.

On the other hand, Baldoni et al. [19] conducted an in-depth and extensive analysis of symbolic execution. Their survey covers the foundational principles, recent advanced techniques, common challenges, and existing solutions. This work serves as a comprehensive resource on the evolution and current state of symbolic execution research.

This research has led to the development of numerous engines and tools that implement symbolic execution for a variety of purposes. Among them, we highlight the following:

- **Driller** [20]: A hybrid vulnerability discovery tool that combines fuzzing with concolic execution. It uses fuzzing to explore common paths quickly and applies symbolic reasoning to reach deeper, less accessible program states.
- **s^2e** (Selective Symbolic Execution) [21]: A symbolic execution platform designed for large-scale systems. It enables the analysis of full software stacks, including operating systems and drivers, by symbolically executing only relevant execution paths.
- **Symbiotic** [22]: A verification framework that integrates symbolic execution with static analysis techniques. It is primarily used to detect memory safety issues in C programs, such as use-after-free errors, invalid dereferences, and memory leaks.

2.5.3 Symbolic Execution for Malware Analysis

While symbolic execution has been extensively explored in the contexts of program testing and formal verification, its direct application to malware analysis remains comparatively limited.

The work of Yan Shoshitaishvili et al. [10] is a key contribution to the field of symbolic execution for binary analysis and malware analysis. In their paper, they emphasize the importance of analyzing binaries and demonstrate how symbolic execution can support tasks such as vulnerability discovery and exploit generation. This effort led to the development of `angr`, a modular and open-source binary analysis framework that integrates and systematizes many state-of-the-art techniques.

Building upon their earlier work, Baldoni et al. [23] published a paper detailing their method for applying `angr` to the analysis of Remote Access Trojans (RATs), providing insights into the internal logic and behavioral structure of these types of malware.

In addition, as previously mentioned, the `SEMA` framework itself is built on top of `angr`, leveraging its binary analysis and symbolic execution capabilities to enable malware analysis at scale.

2.5.4 SEMA-Related Work

As of now, `SEMA` remains the only open-source symbolic execution framework specifically tailored for malware behavior analysis. However, academic literature on `SEMA` remains limited, with the primary reference being the original work by Bertrand Van Ouytsel et al. [4], which introduced the framework.

The only currently available study is that of S. Lucca et al. [24], along with her associated master's thesis [6], which extended `SEMA` to support complete analysis of some Remote Access Trojan (RAT) malware.

In parallel to our work, several other students are also actively contributing to the development of `SEMA`. Notably, T. Bouvencourt is working on the implementation of concolic execution within the framework, while T. Grignard is investigating evasion techniques used by malware. Both of these studies are expected to be released concurrently with this one.

2.6 Summary

In this chapter, we presented the various tools used throughout our work, along with the research relevant to our study.

As outlined in the related work section, while symbolic execution is a well-established technique in software analysis, its application to malware analysis remains comparatively underexplored, with a rather limited amount of tools available.

In the following chapter, we will primarily rely on static analysis—using Ghidra for malware inspection and RetDec for data extraction to support result comparison—as well as symbolic analysis through the `SEMA` toolchain.

Chapter 3

Gonnacry

GonnaCry is an open-source Linux ransomware developed by Tarcisio Marinho [1], with its source code publicly available on GitHub [2]. It was originally created as an academic project aimed at studying ransomware behavior and raising awareness about its risks.

The project currently contains two distinct versions: an older version written in C, and a newer version implemented in Python. These versions follow completely different workflows, meaning that analyzing one provides no insight into the functionality of the other. The GitHub repository includes pre-compiled binaries for both versions. Notably, while the older version retains its symbol information, the newer version has its symbols stripped. As a result, during static analysis using tools such as `Ghidra` or `RetDec`, the function names are visible in the older version but not in the newer one. Additionally, the newer implementation relies on two separate binaries, which further complicates the analysis process compared to the single-binary design of the original version.

For these reasons—combined with the availability of an additional sample of the older version on MalwareBazaar [3]—this section will focus exclusively on the analysis of the older C-based version.

It should also be noted that in the current version of SEMA [4], available on the `production` branch on GitHub, there is no specific implementation targeting ransomware families. However, in the branch named `development ransomware`, some initial work related to ransomware analysis has been introduced. Since this branch is based on a completely different version of SEMA and contains multiple issues—which will be addressed later in this work—additional effort is required to integrate the developments from the `development ransomware` branch into the main `production` branch.

Our analysis is structured into four main parts,. In the first subsection 3.1, we provide an overview of the ransomware's executions phases. Next, in subsection 3.2, we delve into the challenges encountered during the SEMA execution of the ransomware and the solutions applied. Following that, subsection 3.3 presents the results obtained from symbolic

[1] `https://github.com/tarcisio-marinho`

[2] `https://github.com/tarcisio-marinho/GonnaCry/tree/master`

[3] SHA256: `f5de75a6db591fe6bb6b656aa1dcfc8f7fe0686869c34192bfa4ec092554a4ac`

[4] commit: `a294df696a20ad3e3872d8c5ddd63feb842b3492`

execution. Finally, we conclude our analysis in subsection 3.4.

3.1 Functional Overview

We can separate the workflow of *GonnaCry* into four main phases:

User Data Collection

In this phase, GonnaCry scans the user's system to gather various pieces of information. It primarily retrieves paths to key directories such as `Desktop`, `Home`, `Media`, and `Trash`. Additionally, it records the username of the logged-in user, which is specifically used to locate the `Media` folder. The other gathered information is used in the subsequent phases of the attack.

Encryption Target Identification

In this second phase, GonnaCry uses the previously collected directory paths—excluding the `Desktop` path—as targets. It scans the contents of these directories and collects the paths of all files found. All files identified in this step are stored and will be encrypted in the next phase.

Encryption

During this phase, GonnaCry encrypts the collected files using the `EVP_Encrypt` functions from the OpenSSL library [25]. After completing the encryption, the ransomware generates a file containing the encryption keys and other relevant decryption data. This file is then saved to the user's `Desktop`.

Cleanup

In the final phase, GonnaCry deallocates all memory used during execution. Lists are emptied and destroyed, and all variables are cleaned up to ensure that no residual traces remain in memory.

External Calls

Having identified the different phases of the ransomware, we can now focus on the analysis. To support the symbolic execution, we next perform static analysis on the binary to determine the number of external calls made during each phase of execution. This metric allows us to compare the findings from static analysis with those obtained through symbolic execution, providing insight into the completeness and accuracy of the analysis performed by SEMA. It also serves as a useful indicator of the current progress and limitations of SEMA's analysis. To achieve this, we use RetDec to disassemble the binary into a `.dsm` file. A simple Python script is then employed to extract all external calls that could be invoked

during execution. Using this technique, we can create the table 3.1 where we can see for each phases the number of external calls and a representative subset of the calls found in that phase.

Table 3.1: Estimated Number of External System Calls per Phase Based on Static Analysis

Step	**Number of External Calls**	**Representative System Calls**
User Data Collection	24	`getuid`, `getpwuid`, `strcpy`, ...
Encryption Target Identification	32	`strcpy`, `opendir`, `strlen`, `strcmp`, `readdir`, ...
Encryption	69	`EVP_...`[5], `fopen`, `fclose`, `strlen`, `fwrite`, `fread`, ...
Cleaning	6	`free`
Total	**131**	

3.2 Sema execution

In this section, we analyze the main challenges encountered during the execution of GonnaCry using SEMA. We identify three primary issues, which we will examine in detail throughout the remainder of this subsection:

- Faulty `SimProcedure` implementations.
- Limitations of symbolic execution, including issues with path coverage.
- Missing `SimProcedure` implementations required for complete analysis.

3.2.1 Faulty `SimProcedure`

Several existing `SimProcedure` implementations in the current version of SEMA[6] are faulty. These issues can lead to incorrect analysis results or even cause execution to crash. The problem is further exacerbated by inconsistencies and outdated implementations in the `development-ransomware` (DR) branch, which we merged into the main branch.

In this subsection, we analyze several of these faulty `SimProcedure` implementations, explaining their root causes and their impact on symbolic execution.

[5]There are six different system calls that start with `EVP_`.
[6]commit: a294df696a20ad3e3872d8c5ddd63feb842b3492

`getpwuid`

This external call is typically used to retrieve a `passwd` struct based on a user ID (UID). In GonnaCry, it is used to obtain the username of the logged-in user. However, due to a bug in the main branch's implementation, SEMA crashes when it attempts to execute this `SimProcedure`. Fortunately, the DR branch contains a corrected implementation, which we adopted for our analysis.

`fopen`

This function is used to open files for reading or writing. In GonnaCry, it is invoked to read and encrypt file contents. Like `getpwuid`, SEMA crashes when encountering this call. The root cause is a compatibility issue between the version of `angr` used in the main branch (v9.2.21) and the version assumed by the `SimProcedure` from the DR branch. The `SimProcedure` relies on a function not present in the version of `angr` we are using, leading to a crash. To address this issue, we removed the function and introduced a simplified workaround to replicate its intended behavior.

`strtok`

This function splits a string into multiple tokens using a delimiter in a loop. In GonnaCry, it is used to process a string containing a list of file extensions for filtering purposes. The implementation from the DR branch contains several bugs. It returns only the first token, and even that is parsed incorrectly due to invisible whitespace characters. Although this does not crash SEMA, it leads to incorrect outputs in later calls, ultimately producing flawed analysis results.

These examples illustrate how faulty or outdated `SimProcedure` implementations can impair the analysis even more severely than missing ones. In some cases—such as with `getpwuid` or `fopen`—they cause visible crashes. However, in other cases, like `strtok`, they silently produce incorrect or incomplete results, which are often significantly harder to detect and debug.

For example, in the case of `strtok`, the issue was initially detected through incorrect return values from another `SimProcedure`, `strcmp`, which is used to compare two strings. Due to unexpected invisible whitespace, `strcmp` never returned the expected result. At first, we suspected that the issue was in `strcmp`, but after tracing the strings it received, we discovered that the actual source of the problem was an improperly implemented `strtok`.

These three cases are not the only faulty `SimProcedure` implementations we encountered. The following table lists all the functions with faulty implementations that we identified and had to modify during our analysis:

Table 3.2: Summary of Faulty `SimProcedure` Implementations and Fixes

Function	Effect on Execution	Source of Bug	Implementation Source	Fix Applied
`getpwuid`	Execution crash	Incorrect implementation	Main branch	Used fixed version from DR branch
`fopen`	Execution crash	Called non-existent function in used `angr` version	DR branch	Deleted invalid function, manually reimplemented logic
`strtok`	Incorrect parsing, faulty output	Logic not correctly or fully implemented	DR branch	Almost complete reimplementation to match intended behavior
`opendir`	Execution crash	Badly commented code blocking correct execution	DR branch	Recovered older, functional implementation without dead code
`readdir`	Execution crash	Badly commented code blocking correct execution	DR branch	Recovered older, functional implementation without dead code

3.2.2 Symbolic execution limitation

While symbolic execution is a powerful analysis technique, it suffers from specific limitations that must be addressed to enable complete and accurate program analysis. In this section, we focus on two major issues that arose during the execution of GonnaCry.

The first issue is related to the simulation of certain behaviors, which can be challenging to emulate accurately in a symbolic context. Improper simulation may lead to incorrect or misleading results. The second issue concerns path coverage—situations where, for various reasons, some execution paths are never explored. These limitations can significantly hinder the quality and completeness of the analysis.

Simulation Behaviors

In the previous section, we briefly mentioned the `opendir` and `readdir` calls, which failed to work correctly due to the presence of dead code in their corresponding `SimProcedure` implementations. This dead code stemmed from an unresolved question: how should the file system be simulated during symbolic execution?

One option is to allow SEMA to access the real file system, which could result in more accurate analysis but introduces significant security risks, as symbolic execution might operate on real and potentially sensitive system data. The alternative is to simulate the file system entirely within the analysis environment. In the `development-ransomware` (DR) branch, both approaches were attempted—but both were left within the same `SimProcedure`, resulting in dead code and conflicts that ultimately broke the implementation.

During our experiments, we observed that allowing SEMA to access the real file system often resulted in crashes and unpredictable behavior, due both to limitations within the framework and the inherent security risks. To mitigate these issues, we adopted a safer approach: simulating a Linux file system using a custom plugin, `PluginLinuxSystem`.

This plugin constructs a minimal symbolic Linux file system, simulating common user directories such as `Trash`, `Media`, `User`, and `Desktop`, along with their hierarchical structure. It populates these folders with symbolic files using common extensions (e.g., `.doc`, `.txt`) and maps them into memory as symbolic buffers. This setup enables Linux ransomware samples—such as `GonnaCry`—to interact meaningfully with the environment during symbolic execution.

While this solution significantly improved the stability and safety of the analysis process, it came at the cost of reduced completeness due to the simplified nature of the simulated file system.

Path Coverage

Like many programs, GonnaCry includes fail-safe mechanisms—code paths that are executed only when earlier operations fail. In our case, because the initial operations succeeded, these fail-safe paths were never triggered, despite being correctly implemented. This is a common issue in malware analysis, particularly in dynamic analysis, where such structures are often used as evasion techniques to hide malicious behavior from analysis tools.

In `GonnaCry`, the fail-safe mechanism appears to be legitimate. Nevertheless, to ensure complete path coverage, we aimed to analyze these alternative execution paths as well. To achieve this, we intercepted the function responsible for enabling the fail-safe mechanism and modified its return value, thereby forcing execution along both the normal and alternative branches.

Fortunately, since the binary retains its symbol information, we were able to identify the function responsible for the verification, named `is_paths`. This information is particularly useful, as having the symbol associated with the function allows us to easily implement a corresponding SimProcedure to intercept and control its behavior.

In cases where the binary does not preserve symbol information, we would have had to

rely on a more sample-specific hook. Hooks and their implementation strategies will be discussed in greater detail in Chapter 4.

However, intercepting a function introduces a secondary issue: the external calls originally present within the function are no longer executed and therefore do not appear in the execution trace. To address this limitation, we developed a new plugin called `PluginCallReplace`.

This plugin is invoked at the end of the execution of SEMA. It detects function hooks and reinserts the missing external calls using predefined, hardcoded values. For example, in our case, the plugin removes the call to the `is_paths` function and replaces it with the external calls originally present within that function. Specifically, we configure the plugin to insert `opendir` and `closedir` calls, which are normally executed inside `is_paths`.

As a trade-off, the addresses of these reinserted calls include the keyword `replace` to indicate that they do not reflect the actual binary addresses. Instead, they inherit the address of the replaced function, which may affect precise address tracking.

These two examples highlight the primary challenges we encountered with symbolic execution during the analysis of GonnaCry. We believe there are no perfect solutions to these issues—each of our proposed workarounds involves trade-offs that we consider acceptable within our analysis framework.

Furthermore, as we will explore in chapter 4, additional challenges may arise when analyzing other types of ransomware.

3.2.3 Missing `SimProcedure`

In SEMA, when an external call is encountered without an associated `SimProcedure`, that call is omitted from the execution trace. This means the call is absent from both the generated SCDG and the corresponding JSON output. Furthermore, as explained in Chapter 2, symbolic execution cannot simulate the behavior of such calls, since it lacks information about their internal effects. Therefore, it is crucial that all relevant external calls are matched with appropriate `SimProcedure` implementations.

To perform a complete analysis of GonnaCry, we had to implement several custom `SimProcedure` functions, as shown in Table 3.3. This table also highlights the fact that the complexity of a `SimProcedure` can vary greatly. Because we are conducting symbolic analysis, much of the internal behavior of certain functions does not need to be simulated in detail.

Table 3.3: Summary of Custom `SimProcedure` Implementations and Outputs

Function	**Implementation Type**	**Return Value / Output**
`EVP_bf_cbc`	Empty	Always returns 1
`EVP_CIPHER_CTX_cleanup`	Empty	Always returns 1
`EVP_CIPHER_CTX_init`	Empty	Always returns 1
`EVP_CIPHER_CTX_new`	Empty	Always returns 1
`EVP_CIPHER_CTX_reset`	Empty	Always returns 1
`EVP_CIPHER_CTX_set_key_length`	Empty	Always returns 1
`EVP_CipherInit_ex`	Empty	Always returns 1
`EVP_CipherUpdate`	Some internal logic	Returns both 0 and 1 to enable path exploration
`EVP_CipherFinal_ex`	Some internal logic	Returns both 0 and 1 to enable path exploration
`fread`	Special case (see associated paragraph)	Random value to avoid infinite loop
`fwrite`	Complete logic	Returns real written byte count
`rewind`	Complete logic	Returns 0 if successful

For instance, many of the OpenSSL `EVP` functions used to create cipher environments or encrypt data do not require full simulation, since our goal is not to perform real encryption. As a result, many of these `SimProcedure` implementations are effectively empty and return null values.

However, certain functions such as `EVP_CipherUpdate` and `EVP_CipherFinal_ex` perform operations that influence execution flow and must therefore be partially simulated. Additionally, for calls that influence conditional behavior—such as `is_path` mentioned earlier—it is important to return values that allow multiple execution paths to be explored. In such cases, Claripy can be used to create symbolic variables that can take on multiple possible values.

In the same table, we also highlight a special case: `fread`, which requires a more detailed explanation.

`fread`

This function is used to read from a file and returns the number of characters read. However, due to limitations in the Linux file system plugin, even with a correct `fread` implementation, the function always returns 0, which leads to infinite loops in GonnaCry's execution. This behavior can block or significantly slow down the analysis.

While one option would be to return multiple symbolic values to explore various

outcomes, this approach does not solve the infinite loop issue. The only practical solution we found was to return a random value. This not only avoided infinite loops but also enabled full path exploration. Although this method is not ideal, it proved effective for our analysis purposes.

However, this approach is specifically tailored for the `GonnaCry` sample. For this reason, we retained the original implementation within the `SimProcedure`, simply commenting it out to allow for easy reactivation in other contexts.

Normally, the `EVP_Cipher(...)` function returns `1` on success and `0` on failure. In our case, `GonnaCry` checks only the return values of `EVP_CipherUpdate` and `EVP_CipherUpdate_ex`. For this reason, these two calls are configured to return a BitVector symbolic(BVS) that can simultaneously represent both `0` and `1`, enabling exploration of all possible execution paths.

On the other hand, `EVP_bf_cbc` typically returns an `EVP_CIPHER` structure, which is used by subsequent `EVP_Cipher(...)` calls. However, since these calls are empty in our context, the structure is not required. To simplify the implementation, we returned `1` in place of the full structure. This approach reduced complexity and improved execution efficiency without compromising the objectives of the analysis.

3.3 Call Coverage Analysis

To analyze the effectiveness of SEMA, we compared the number of external calls it produced with those identified through static analysis (Table 3.1). This comparison was conducted across three distinct stages of the implementation:

- At the beginning, using only the `SimProcedure` implementations already present in SEMA.
- After merging the `development-ransomware` (DR) branch into the main branch.
- At the end, after resolving all major issues and completing the missing `SimProcedure` implementations.

In the first two stages, we made minimal changes to the existing `SimProcedure` implementations, focusing mainly on preventing crashes during execution. The results at each stage are presented in Table 3.4, which compares the number of external calls observed during symbolic execution with those obtained from static analysis.

Table 3.4: Evolution of External Call Coverage During Symbolic Execution

Step	**Static Call**	**First Execution**		**After Merge**		**Final Execution**	
		Unique	Total	Unique	Total	Unique	Total
1	24	20	20	20	20	24	26
2	32	2	6	20	51	32	57
3	69	0	0	0	0	69	120
4	6	5	5	5	5	6	9
Total	**131**	**27**	**31**	**45**	**76**	**131**	**212**

In this table, each metric is divided into two columns: *unique* and *total*. This distinction reflects how external calls are recorded differently in static analysis versus symbolic execution.

The static analysis script registers a call only once per binary address, even if the binary address is visited multiple times—for instance, when a function containing an external call is invoked repeatedly throughout the program. In contrast, `SEMA` logs all calls observed during symbolic execution. Although it includes filtering mechanisms to avoid duplicating identical calls from the same execution context (e.g., same arguments and control flow), it still records a call multiple times if it is encountered under different execution contexts.

Consequently, if a function is called in different contexts during execution, static analysis report the external call inside it only once, while symbolic execution may record it multiple times for each call of the function. The distinction between *unique* and *total* helps provide a more comprehensive and accurate comparison between the results of static and symbolic analyzes.

Figure 3.5 visualizes the overall progression in system call coverage across these three stages.

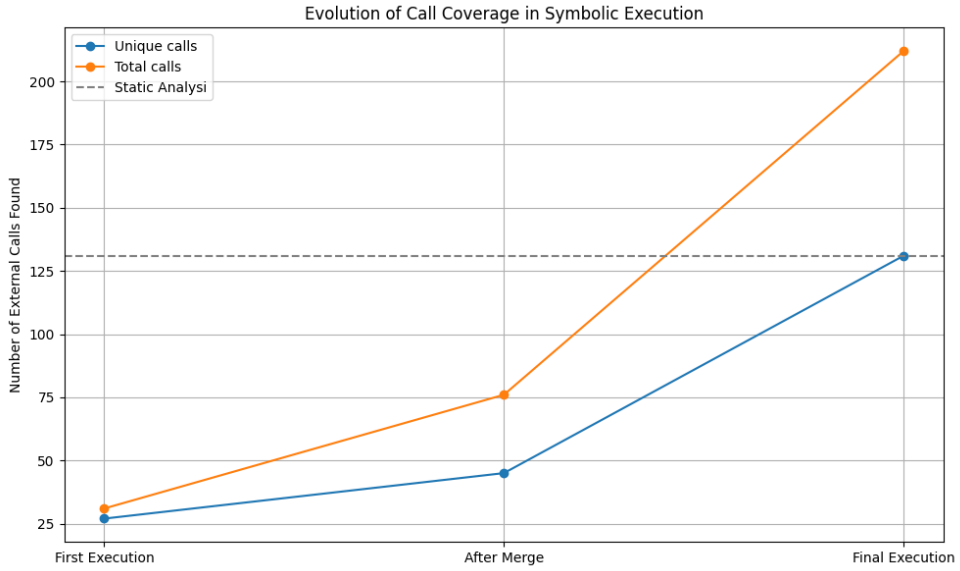

Figure 3.5: Call Coverage Evolution

It should be noted that there is no difference in the results between the sample obtained from the official GitHub repository and the one retrieved from MalwareBazaar.

3.4 Conclusion

This section presented the implementation of a complete analysis of the GonnaCry ransomware using SEMA-SCDG. Our primary objective was to ensure a comprehensive and accurate analysis of the ransomware through symbolic execution within the SEMA toolchain.

Through multiple improvements—including the integration of the `development-ransomware` branch—we successfully achieved full execution of the sample. This resulted in the identification of 212 total external calls, exceeding the number found through static analysis by 61%.

However, this progress is not without trade-offs. Certain implementation decisions have resulted in the loss of some information. It would be valuable in future work to further investigate and extend the `PluginLinuxSystem`, enhancing its capabilities to enable more comprehensive simulation of the Linux file system. This would be especially beneficial for analyzing larger and more complex ransomware samples.

Chapter 4

WannaCry

`WannaCry` is a piece of malware that targets Windows-based systems. It functions both as ransomware and as a worm—encrypting files while simultaneously propagating itself to other vulnerable machines on the same network. The initial `WannaCry` outbreak lasted only 7 hours and 19 minutes before cybersecurity researcher Marcus Hutchins discovered a kill switch embedded in the malware. By registering a specific domain name that the malware was programmed to contact, he was able to halt its global spread entirely [26, 27].

Despite its brief active period, `WannaCry` had a significant global impact, infecting an estimated 200,000 computers across 150 countries. The ransomware leveraged a Windows vulnerability known as *EternalBlue*, along with a post-exploitation tool called *DoublePulsar*. Both were developed by the NSA for internal use and later leaked and weaponized. Responsibility for the attack has been widely attributed to the Lazarus Group, a state-sponsored hacking organization linked to North Korea [28, 29].

EternalBlue is an exploit that targets a vulnerability in the SMBv1 protocol on Windows systems. It allows attackers to gain remote access to machines over a network. Although Microsoft released a patch in March 2017 following the leak of the exploit, many systems remained unpatched when the `WannaCry` attack began in April 2017 [30].

On the other hand, *DoublePulsar* is a tool used in conjunction with *EternalBlue* to inject a backdoor into compromised systems, enabling persistent remote control [30].

The kill switch embedded in the malware is believed to be the result of a poorly implemented evasion technique. As shown in Figure 4.1, before executing its core ransomware functionality (function `FUN_00408090` on the Figure), the malware attempts to contact a hardcoded domain. If the domain responds, the malware halts execution and becomes inactive. This type of behavior is commonly used to detect execution in sandboxed or virtualized environments, typically by checking for unreachable or randomized domains. In the case of WannaCry, however, the domain was fixed and unregistered, making it possible for Hutchins to neutralize the malware's spread by registering the domain.

The WannaCry ransomware operates through multiple layers of components [31].

The first layer, known as *launcher.dll*, is a trojan that leverages the EternalBlue exploit and the DoublePulsar backdoor. Its role is to inject the next stage of the malware into the target system. Due to the limited information available and its out-of-scope nature, this

```
uVar1 = InternetOpenA(0,1);
iVar2 = InternetOpenUrlA(uVar1,&uStack_64,0,0,0x84000000,0);
if (iVar2 == 0) {
  InternetCloseHandle(uVar1);
  InternetCloseHandle(0);
  FUN_00408090();
  return 0;
}
InternetCloseHandle(uVar1);
InternetCloseHandle(iVar2);
return 0;
```

Figure 4.1: The WannaCry kill switch mechanism: execution stops if the hardcoded domain is reachable

component will not be studied in this work.

The injected executable, `mssecsvc.exe`, functions primarily as a worm and a dropper. It is a self-contained binary that exploits Windows vulnerabilities—most notably EternalBlue—to gain access to target systems. Once a system is compromised, it unpack a second payload (`tasksche.exe`) and propagates itself to other vulnerable machines on the network. This component also contains the well-known kill switch mechanism.

The `tasksche.exe` file is another dropper. It hides its presence on the system and creates several subcomponents, such as `t.wnry` and `c.wnry`. These files are binary data rather than executables and are used to store configuration and encrypted payloads. As such, they will not be analyzed in detail in this study.

The extracted payload includes a DLL named `kbdlv`, which is the core ransomware component. This DLL is responsible for encrypting and decrypting files, as well as displaying the ransom note to the user.

In the following analysis, we examine samples of `mssecsvc.exe`, `tasksche.exe`, and `kbdlv`. We then extend our investigation to additional samples obtained from MalwareBazaar, before concluding our analysis of WannaCry.

4.1 `mssecsvc.exe`

In this subsection, we analyze the `mssecsvc.exe`[1] component of WannaCry.

We begin by examining the functionality of this component and identifying the number of external calls revealed through static analysis. We then discuss the challenges encountered during the implementation efforts required to enable SEMA to fully explore this sample. Following that, we compare the results obtained through symbolic execution and highlight specific observations made possible only via this approach.

This component also includes worm-like propagation behavior, which lies outside the

[1]SHA256 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

scope of our current research. Nevertheless, we conclude the subsection with a brief discussion of our insights into this functionality.

4.1.1 Functional Overview

This component operates in three main steps [31]:

Kill Switch

As mentioned earlier, the first step executed by the dropper is the kill switch mechanism, shown in Figure 4.1. Since this functionality has already been discussed in detail, we will not elaborate further here.

Dropper Phase

In this initial functional phase, the dropper performs two key actions. First, it creates a new Windows service named `mssecsvc2.0`, which is later used to propagate the malware across the network. Second, it extracts a file named `tasksche.exe` from its internal resources and drops it onto the system. This file constitutes the second stage of the WannaCry attack, which we will analyze in the section 4.2.

Propagation Phase

In the final stage, the binary launches the `mssecsvc2.0` service, initiating the spread of the ransomware to other vulnerable machines within the network. While it is technically possible to analyze the `mssecsvc2.0` service using SEMA or static analysis, we exclude it from our scope. This component handles network propagation via the EternalBlue exploit, which lies outside the core ransomware functionality we aim to study.

Using `RetDec` and a filtering script similar to the one applied for `GonnaCry`, we estimated the number of external calls made during each of the three phases of the component's execution. The results are summarized in Table 4.2.

In total, we identified 38 external calls, including 6 obfuscated calls. These obfuscated calls do not directly reference known external functions but instead rely on indirect techniques, such as targeting general-purpose registers (e.g., `EAX`) or memory addresses (e.g., `DWORD` values), which makes them more difficult to trace during static analysis.

As a result, these obfuscated calls are excluded from the static analysis but will be discussed in Subsection 4.1.3, leaving a total of 32 clearly identifiable external calls, distributed as follows:

4.1.2 SEMA Execution

During our analysis of the WannaCry `mssecsvc.exe` Component, we encountered two major issues and identified one missing `SimProcedure`. The problems were related to the

Table 4.2: Estimated Number of External Calls per `mssecsvc.exe` Component Phase (Static Analysis)

Phase	**Number of External Calls**	**Representative Calls**
Kill Switch	6	`InternetOpenA`, `InternetOpenUrlA`, ...
Dropper Phase	20	`CreateServiceA`, `FindResourceA`, `GetProcAddress`, `sprintf`, ...
Propagation Phase	6	`OpenServiceA`, `StartServiceCtrlDispatcherA`, ...
Total	**32**	

VEX engine used by `angr` and the resource plugin integrated into SEMA.

VEX Engine

This issue originates from within `angr` itself, making it particularly difficult to isolate and resolve. The error occurs when `angr` attempts to access the address of the next instruction under the assumption that it is a constant value. In reality, the address is stored in a temporary register (`RdTmp`), which lacks the `con` attribute typically used to retrieve constant values. As a result, an `AttributeError` is raised during execution, causing a crash in the VEX engine.

Identifying the root cause of this issue is complex. It may stem from prior instructions, incorrect or incomplete `SimProcedure` implementations within SEMA, quirks or anomalies in the malware binary itself, or inherent limitations in `angr`'s internal handling of VEX intermediate representation.

In our case, the crash was triggered by the `__initterm` call found in the installation routines common to many Windows executables. Since this call is not essential for our analysis and does not significantly influence the behavior of the component, we bypassed the issue by modifying its associated `SimProcedure` to return immediately, effectively skipping its execution.

Resource Plugin

The second issue involved the `FindResourceA` call, which is implemented using the resource plugin in SEMA. In our experiments, the plugin failed to function correctly, causing the `SimProcedure` to enter an infinite loop. After extensive debugging, we were unable to find a practical fix. As a workaround, we replaced the output of this `SimProcedure` with a symbolic variable, allowing the analysis to proceed.

Although these workarounds are not ideal, they allowed SEMA to explore the complete execution path of the component. However, this comes at a cost: under normal execution, the component should create a new executable named `tasksche.exe`. With our current implementation, the file is created, but in a corrupted state due to skipped or incomplete procedures.

Missing `SimProcedure`

The only missing `SimProcedure` was for the `__p___argc` call, which is typically used to retrieve a pointer to the number of program arguments. In our case, this value is used to determine whether to execute the full dropper phase or proceed directly to the propagation phase.

Since we aim to explore both execution paths, we returned a pointer to a symbolic variable. We chose to leave this variable unconstrained to maintain generality and simplify the analysis. Constraints can easily be added later if needed for the analysis of other malware samples.

4.1.3 Result Analysis

As shown in Table 4.2, the component relies on approximately 32 external calls. Our symbolic execution analysis with SEMA identified a total of 66 external calls. However, 11 of these originate from standard Windows installation routines, which are not directly related to the functional behavior of the component.

Excluding these, SEMA successfully detected 55 relevant external calls, distributed as follows:

Table 4.3: Comparison of Static and SEMA-Observed External Calls per Phase

Phase	**Static Call**	**Unique Calls (SEMA)**	**Total Calls (SEMA)**
Kill Switch	6	6	9
Dropper Phase	20	26	34
Propagation Phase	6	6	12
Total	**32**	**38**	**55**

As observed previously, the total number of external calls identified by SEMA is higher than that found through static analysis. This discrepancy arises because SEMA logs each instance a call is visited, even if it occurs at the same location multiple times.

Another noteworthy observation is that during the Dropper phase, SEMA identified six additional external calls that were not clearly visible through static analysis. These are unique calls—distinct from repeated entries—and demonstrate SEMA's ability to reveal hidden or obfuscated behavior within the binary that traditional static methods may overlook.

For example, at addresses `0x407EF7` and `0x407F02`, the decompiler shows two type casts, as illustrated in Figure 4.4. However, the disassembler reveals two calls to a `DWORD`-type value, shown in Figure 4.5. SEMA, through symbolic execution, was able to resolve these and correctly identify them as `CloseHandle` calls

```
if (iVar4 != 0) {
  (*DAT_0043144c)(uVar12);
  (*DAT_0043144c)(uVar13);
}
```

Figure 4.4: Decompiler view showing type casts at addresses `0x407EF7` and `0x407F02`

```
00407ef6 50              PUSH    EAX
00407ef7 ff 15 4c        CALL    dword ptr [DAT_0043144c]
         14 43 00
00407efd 8b 4c 24 14     MOV     ECX,dword ptr [ESP + 0x14]
00407f01 51              PUSH    ECX
00407f02 ff 15 4c        CALL    dword ptr [DAT_0043144c]
         14 43 00
```

Figure 4.5: Disassembler view showing hidden function calls interpreted as `DWORD` types

The following table lists the external calls identified by SEMA, their corresponding addresses in the binary, and how they are represented statically using Ghidra's decompiler. In each case, the disassembler shows the call as an invocation through a `DWORD`-type data pointer, which obscures the function's identity in static analysis.

Table 4.6: SEMA-Observed Calls with Corresponding Binary Address and Ghidra Representation

Call (SEMA)	Binary Address	Ghidra Representation
`CreateFileA`	0x407e43	`uStack_290 = 0x407e49;`
`WriteFile`	0x407e61	`(*DAT_00431460)(iVar4, uVar13, DVar3, &stack0xfffffd84);`
`CloseHandle`	0x407e68	`(*DAT_0043144c)(iVar4);`
`CreateProcessA`	0x407ee8	`iVar4 = (*DAT_00431478)(0, acStack_23c, 0, 0, 0, 0x8000000, 0, 0, &stack0xfffffd80, &uStack_290);`
`CloseHandle`	0x407ef7	`(*DAT_0043144c)(uVar12);`
`CloseHandle`	0x407f02	`(*DAT_0043144c)(uVar13);`

This pattern results from the use of the `GetProcAddress` function, which dynamically resolves the addresses of API functions at runtime. As a consequence, these calls appear in the binary as indirect calls to memory locations (i.e., pointers to `DWORD` values), effectively obfuscating their identities and making them harder to detect through static analysis.

4.1.4 Worm Insights

As previously mentioned, the worm functionality is accessible through both static and symbolic analysis, though it is somewhat obscured and can be easily overlooked.

In the case of static analysis, the worm can be reached through the final function call in the binary:

```
}
SStack_10.lpServiceName = s_mssecsvc2.0_004312fc;
SStack_10.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)&LAB_00408000;
uStack_8 = 0;
uStack_4 = 0;
StartServiceCtrlDispatcherA(&SStack_10);
return;
```

Figure 4.7: Final call of the binary with worm entry point

In this code snippet, we observe the last function call of the binary and its argument. By following the `lpServiceProc` argument (shown as `LAB_00408000` in the figure), we can access what appears to be the worm-related code.

To explore this functionality with SEMA, we must modify the `StartServiceCtrlDispatcherA SimProcedure` so that execution jumps to the worm's entry point. This jump logic has already been included in the SimProcedure implementation, and can be activated by simply uncommenting the relevant line as needed.

4.2 `tasksche.exe`

In this subsection, we analyze a sample of the `tasksche.exe` component [2].

We follow the same structure used in the analysis of the `mssecsvc.exe` component. First, we examine it's behavior and estimate the number of external calls identified through static analysis. Next, we discuss the challenges encountered during symbolic execution. Finally, we compare the results obtained from symbolic analysis with those derived from static analysis.

4.2.1 Functional Overview

The operations of `tasksche.exe` can be divided into four main steps [32].

Hidden Copy

At first, `tasksche.exe` generates a pseudo-random string derived from the computer's name. If the malware detects that it was executed with the `/i` argument, this string is used to create a hidden directory. A copy of the component is then placed in this directory and executed without the `/i` argument, allowing the malware to continue running stealthily in the background.

[2] SHA256: `ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa`

Resource Extraction

It will then extracts embedded resources to create several files, including `c.wnry` and `t.wnry`. The file `c.wnry` contains configuration data, including a Bitcoin address used for tracking ransom payments.

RSA Key Decryption

WannaCry uses an embedded RSA key to decrypt a portion of the `t.wnry` file. This process yields an AES key, which is required to decrypt the actual ransomware payload.

DLL Extraction

Using the decrypted AES key, WannaCry decrypts and loads the `kbdlv` DLL embedded within `t.wnry`. This DLL is responsible for continuing the encryption process and for displaying the ransom note to the victim.

Due to the limitations of SEMA—primarily the non-functional resource plugin—and the constraints of our analysis, which did not achieve full path coverage, we were unable to simulate execution paths that lead to the creation of any of the referenced files. Nonetheless, we were able to obtain the `kbdlv` file from an external source for use in the next section of this chapter [3].

By using RetDec and applying the same filtering method as with previous samples, we identified a total of 280 external calls in the sample. Of these, 11 originate from the Windows initialization process, and 54 are obfuscated calls. Excluding these, the remaining 215 calls are distributed as follows:

Table 4.8: Estimated Number of External Calls per Phase (Static Analysis)

Phase	**Number of External Calls**	**Sub-set Calls**
Hidden Copy	48	`GetWindowsDirectoryW`, `CreateDirectoryW`, `GetComputerNameW`, ...
Resource Extraction	87	`FindResourceA`, `_mbsstr`, `memcpy`, ...
RSA Key Decryption	24	`GetProcAddress`, `GlobalAlloc`, `CreateFileA`, ...
DLL Extraction	56	`memcpy`, `HeapAlloc`, `GetModuleHandleA`, ...
Total	**215**	

[3] `https://github.com/ghidraninja/ReversingWannaCry`

4.2.2 SEMA Execution

During our analysis, we encountered multiple issues and missing `SimProcedure` definitions. These challenges ultimately prevented us from fully analyzing the binary. In this section, we describe the problems we faced, along with some of the solutions we attempted, and explain why full path coverage could not be achieved.

VEX Engine

As with the previous WannaCry sample, we encountered several VEX engine crashes during analysis. As explained earlier, these crashes are difficult to trace due to the complexity of angr's internal logic. However, in this case, we determined that two of the three crashes stemmed from the initialization and destruction of a C++ structure integrated into the binary. These calls appear to trigger undefined behavior in angr's backend.

We resolved the issue by creating three hooks to bypass the specific functions responsible for the crashes. While this solution does result in the loss of internal call information, we mitigated this by using our previously developed post-execution plugin to reinsert the missing calls into the final trace. By combining static and symbolic analysis, we ensured that the reinserted calls reflected realistic behavior. To indicate that these calls were reconstructed after execution, we marked their addresses and other unreliable information in the trace with the keyword `replace_` to signal partial inaccuracy.

Loop-Heavy Function

In the malware's first phase, a pseudo-random string is generated based on the system's information. This is one of the first functions to execute, but it includes multiple heavy loops that dramatically slow down symbolic analysis. To mitigate this issue, we implemented a hook to bypass the function and directly substitute a predefined string, thereby accelerating the analysis without significantly impacting the execution logic.

Resource Plugin

As with the previous binary, we were unable to get the resource plugin to function correctly. Consequently, the `FindResourceA` call failed, even though `__initterm` executed successfully in this case. Normally, `FindResourceA` is used to locate embedded resources within the binary—these resources may include executables, scripts, images, or other data.

In our case, the failure of this call meant that even if we achieved full path coverage and succeeded in triggering the file creation routines mentioned earlier, the resulting outputs would be corrupted, as observed with the first component of WannaCry.

To address this, we applied the same workaround as before: we skipped the problematic call and returned a symbolic value, allowing execution to proceed without halting.

Inconsistencies in Call Resolution and Execution Paths

During our analysis, we observed that certain external calls were handled correctly by SEMA—even in the absence of corresponding `SimProcedure` implementation. These calls produced consistent symbolic behavior and did not disrupt execution flow.

However, when we attempted to define `SimProcedure` for these calls (even as empty placeholders), execution failures occurred. For example, adding a `SimProcedure` for `operator_delete` (also known as `??3@YAXPAX@Z`) led to immediate control flow corruption upon return, resulting in invalid jumps and corrupted execution flow.

Additionally, four other standard calls—`fopen`, `fclose`, `fread`, and `fwrite`—were handled correctly by default. However, when we attempted to override these calls with custom implementations (based on angr's codebase or documentation), execution either failed or entered infinite loops. The origin of their default handling remains unclear; it may involve angr's internal stubs, fallback mechanisms, or legacy behaviors. We ultimately chose not to implement `SimProcedure` for these calls, preserving analysis stability.

Complex Code

As is common with advanced malware, the ransomware's code is highly complex and deliberately difficult to analyze. It makes extensive use of low-level programming techniques such as type casting, pointer arithmetic, and obfuscation strategies. These patterns can pose significant challenges for symbolic execution frameworks such as SEMA.

Figure 4.9 shows a two-line code snippet from the decompiled binary. The intended behavior is relatively clear: retrieve the argument vector using `__p___argv` and compare the second argument to a string (likely `/i`) using `strcmp`. However, due to the multiple levels of casting and pointer manipulation, SEMA is unable to model this behavior correctly.

Our objective was to craft an output in `__p___argv` that, after undergoing the cast and pointer logic, would result in a controllable input to `strcmp`.However, despite multiple approaches and implementation attempts, we were unable to achieve the desired result—SEMA consistently failed to propagate a meaningful, controllable value through this expression.

As a workaround, we introduced a targeted exception inside the `strcmp SimProcedure` to simulate this specific comparison. While not a general-purpose solution, it was sufficient to proceed with our analysis in this case.

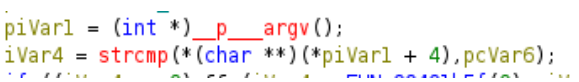

Figure 4.9: Decompiled code showing complex pointer logic involving `__p___argv` and `strcmp`

Unreachable Path

In some cases, we encountered functions that appeared to be unreachable, or whose logic was difficult to interpret. Figure 4.10 shows an example: the `swprintf` function is only called if the `param_3` argument is set. However, analysis in Ghidra reveals that this function is never invoked with a third argument, leaving us uncertain whether this path is truly unreachable or triggered by an obscure condition.

```
if (param_3 != (wchar_t *)0x0) {
  swprintf(param_3,0x40eb88,param_1);
}
```

Figure 4.10: `swprintf` guarded by a parameter condition that is never satisfied during analysis

C++ Structures

As noted in the paragraph on the VEX engine, the ransomware includes a C++ structure that is involved in the final two phases of execution. The presence of this structure significantly increases the complexity of both analysis and interpretation. Our unfamiliarity with C++ semantics, combined with the need to bypass both the construction and destruction routines (due to crashes), leaves us uncertain whether SEMA can correctly analyze these segments of the binary.

Implemented or Modified SimProcedures

In this section, we regroup the different `SimProcedure` files that were implemented or modified for the analysis of the WannaCry ransomware. Instead of listing all procedures in a single long table, we organize them by behavior and implementation strategy to make the design rationale more transparent.

Simple Placeholders These procedures contain no internal logic affecting memory or control flow, but include informative output. Their return values were adapted to enable symbolic execution to proceed without interruption.

- Always return 1: `CreateDirectoryA`, `CreateDirectoryW`, `CloseServiceHandle`, `SetFileTime`, `LocalFileTimeToFileTime`, `SetFileAttributesW`
- Return symbolic 0 or 1: `SetCurrentDirectoryA`, `SetCurrentDirectoryW`
- Return symbolic values: `OpenMutexA`, `GetFileAttributesA`, `GetFileAttributesW`, `__allmul`, `__p_commode`, `__p_fmode`, `GetProcessHeap`
- Return `argv` pointers: `__p__argv`

- Return nothing: `Sleep`, `EnterCriticalSection`, `LeaveCriticalSection`, `CryptReleaseContext`, `DeleteCriticalSection`, `InitializeCriticalSection`, `_CxxThrowException`, `_local_unwind2`, `__setdefaultprecision`
- Always return 0: `__setusermatherr`

Removed for Stability Some `SimProcedure` implementations had to be removed, as their inclusion caused crashes or incorrect behavior during execution.

- `??3@YAXPAX@Z`, `fopen`, `fclose`, `fread`, `fwrite`

Naive Implementations These procedures were implemented with simplified logic. They are not fully accurate, but sufficient for symbolic execution.

- Return true value or symbolic fallback: `_mbsstr`, `_stricmp`
- Return 0 or 1 based on check: `IsBadReadPtr`
- Always return 1: `SystemTimeToFileTime`

Adapted from Existing Implementations These procedures reuse logic from other versions or platforms to reduce development time while maintaining reasonable accuracy.

- Copied from other variant: `GetCurrentDirectoryA` (from `GetCurrentDirectoryW`)
- Adapted from W-version: `GetFileSizeEx`, `GetFullPathNameA`
- Adapted from string-based equivalents: `wcscat`, `wcsrchr`, `swprintf`
- Reuse Linux logic: `strlen`, `strcmp`

Special Cases The following procedures required more involved or custom handling:

Due to our inability to explore the entire ransomware execution path, some of these `SimProcedure` implementations were never executed or tested. As a result, they may contain bugs or imperfections.

Table 4.11: Special-Case SimProcedure Implementations

Function	Type	Behavior / Notes
`wcslen`	Full logic	Fully reimplemented to compute string length correctly
`ReadFile`	Patched	Returns symbolic value (BVS) to avoid blocking behavior
`GetExitCodeProcess`	Fake logic	Always returns `1` to simulate a running process
`GetTempPathW`	Fake logic	Returns length of a predefined fake path
`RegCreateKeyW`	Fake logic (inline output)	Returns `0`, writes a symbolic handle to output buffer
`GlobalAlloc`	Memory allocation	Allocates symbolic memory and returns pointer
`HeapFree`, `GlobalFree`	Memory deallocation	Simulates freeing, returns `0` or `1`

4.2.3 Result Analysis

As shown in Table 4.12, the `tasksche.exe` binary contains approximately 215 unique external calls, based on static analysis. The following table compares these results with the output obtained through symbolic execution with SEMA.

Table 4.12: Estimated and Observed External Calls per Encryptor Phase

Phase	Static Calls	Unique Calls (SEMA)	Total Calls (SEMA)
Hidden Copy	48	35	88
Resource Extraction	87	26	59
RSA Key Decryption	24	23(+3)	37
DLL Extraction	56	6(+7)	15
Total	**215**	**90(+10)**	**199**

The numbers presented in this table exclude post-execution reinjected calls (15 additional calls). Additionally, the results should be interpreted with caution, as certain functions are reused across multiple phases of execution, which may lead to discrepancies in the static call count.

Moreover, in the "Unique Calls" section, numbers shown in parentheses represent calls uniquely identified by SEMA that were not detected through static analysis due to obfuscation techniques.

As shown, the symbolic execution results for this part of WannaCry are relatively limited: only 41% of the external calls identified through static analysis were also observed

during symbolic execution. This reduced coverage is primarily attributable to the three major issues discussed earlier (prior to the part on missing `SimProcedures`).

While problems such as VEX engine crashes and plugin integration were time-consuming, they were ultimately addressed with reasonable success. However, the last set of issues significantly hindered the analysis, slowing down execution and limiting path coverage.

4.3 `kbdlv` DLL

In this subsection, we analyze the `KDBLV` DLL component of WannaCry [4].

Unlike previous samples, this DLL was not obtained from MalwareBazaar but is instead available on GitHub[5] as part of the repository associated with Stacksmashing's Ghidra-based analysis of the ransomware, as presented on YouTube [32].

We maintain the same analytical structure as in prior sections: beginning with a functional overview of the component, followed by a discussion of the challenges encountered during symbolic analysis along with potential solutions or areas for further exploration. We conclude by comparing results obtained through symbolic execution with those derived from static analysis.

4.3.1 Functional Overview

The execution can be separated into three distinct phases:

Initialization

In this phase, the malware performs several initialization routines. These include resolving API calls dynamically (often through function pointers to obfuscate behavior) and checking for or creating a mutex to ensure that only a single instance is running. More importantly, it verifies the presence and content of four specific files: `.pky`, `.eky`, `.res`, and `.dky`. Among these, the `.res` file contains victim-specific metadata that may be used during communication with the attacker. The other three files store cryptographic material: the `.pky` file contains the attacker's public RSA key; the `.dky` file holds the victim's locally generated private RSA key; and the `.eky` file stores the same private key, but encrypted with the attacker's public key. Normally, the `.dky` file is present only after the ransom has been paid.

Thread Creation

During this phase, the ransomware primarily creates five different threads. The first is responsible for writing the current timestamp into the `.res` file. The second checks whether the victim has paid the ransom by monitoring the presence of the `.dky` file. The third

[4] SHA256: `1be0b96d502c268cb40da97a16952d89674a9329cb60bac81a96e01cf7356830`

[5] `https://github.com/ghidraninja/ReversingWannaCry`

continuously scans for newly added drives (such as USB keys) and encrypts their contents. The fourth and fifth threads launch two additional executables: `taskdl.exe`, which deletes temporary files created by WannaCry, and `@WanaDecryptor@.exe`, which serves as the visual interface and facilitates command-and-control (C&C) communication.

Encryption

This final phase implements the core encryption routine. After performing several preliminary checks and setup steps, the malware drops a README file named `@Please_Read_Me@.txt` and creates a shortcut to `@WanaDecryptor@.exe` on the desktop. It then scans and encrypts files located in the Desktop and Documents folders, excluding its own files and critical system files. Following this, it enters a continuous loop in which it encrypts additional files across the system until the ransom is paid. Additionally, it periodically restarts `@WanaDecryptor@.exe` if it has been closed.

Using the same methodology as in previous analyzes—though with a slightly modified filtering script to account for thread-related calls, which were not handled correctly in the original version—we identified a total of 447 unique external calls. Among them, 59 are obfuscated calls, leaving 388 clearly identifiable calls. These calls are distributed as follows:

Table 4.13: Estimated Number of External Calls per Phase (Static Analysis)

Phase	**Number of External Calls**	**Representative Calls**
Initialization	93	`GetProcAddress`, `CryptGetKeyParam`, `WriteFile`, ...
Thread Creation	41	`CreateThread`, `Sleep`, `CloseHandle`, ...
Encryption	254	`GetFileAttributesW`, `_wcsicmp`, `DeleteFileW`, ...
Total	**388**	

It should be noted that these numbers must be taken with an even greater degree of caution than before, for several reasons:

- The code frequently reuses the same functions across different phases. However, when this occurs, calls within such functions are only counted in one phase. This behavior is not new, but it is particularly noticeable in this executable.

- `RetDec` appears to have encountered difficulties disassembling this binary correctly. In particular, the function invoked by one of the threads—responsible for scanning newly connected drives to encrypt—does not appear to have been disassembled at all.
- The binary contains dead code, which our script is not equipped to detect or filter out. As a result, some calls listed in the table may not actually be reachable or executed at runtime.

4.3.2 SEMA Execution

During our analysis, we encountered several issues that significantly impacted our symbolic execution of the ransomware components. In this section, we review the main challenges faced during our analysis. We conclude the subsection with a discussion of the `SimProcedures` we added or modified.

Manual Entry Point Adjustment

Before discussing the major issues encountered during analysis, it is important to highlight a fundamental setup constraint. Since the analyzed binary is a `DLL`, `SEMA` does not automatically execute the correct logical entry point of the malware. By default, symbolic execution begins at the binary's formal entry point, which is not where the core logic resides.

As a result, the entry point had to be manually adjusted to the function that initiates the main malware routine—identified as `TaskStart`—located at address `0x10005ae0` in our sample. Without this redirection, `SEMA` would not explore the meaningful execution paths of the payload.

Loop-Heavy Execution

The code contains numerous loops—some of which execute for a short duration, others for extended periods, and some that are designed to loop almost indefinitely. While these loops pose no issue during normal execution, and many terminate quickly, they present significant challenges in the context of symbolic execution.

On one hand, even short loops are much slower to evaluate under symbolic execution than during native execution. On the other hand, some loops rely on pointer manipulation or global variables to terminate. While this works well in concrete execution—due to correct pointer handling and multi-threaded coordination—these features are not robustly supported by `SEMA`. As seen earlier, `SEMA` occasionally struggles with symbolic pointer resolution, and its current version does not yet support multi-threaded execution, though we discuss this limitation later in more detail.

These problems result in some loops becoming infinite, while even terminating ones incur significant performance costs. Unfortunately, there is no perfect mechanism available to resolve this. Although `angr` includes a `loop_counter` feature that terminates execution paths after excessive looping, we were unable to rely on it. In our case, a significant

portion of the code lies beyond these loops, and truncating the path would prevent us from analyzing essential behavior.

Given our time constraints, our only viable approach was to allow `SEMA` to run through limited loops as-is, while ensuring that problematic loops were never reached—achieved using hooks and other workarounds. Future work could benefit from implementing a mechanism to selectively skip such loops, especially since many of them serve no real purpose other than acting as delays or obfuscating the analysis.

Thread Implementation

As previously mentioned, the original malware relies on multi-threaded execution. While `SEMA` includes a plugin named `PluginThread`—designed specifically to manage multi-threaded behaviors—and contains code intended to support such programs, this functionality is currently non-operational in the latest version of the toolchain.

To work around this limitation, we modified the implementation of the `CreateThread` call. However, this workaround introduced additional issues: the functions executed by these threads often contain loops that only terminate when a global variable is modified. In the context of symbolic execution, this results in the analysis getting stuck for extended periods within these loops.

Due to these complications, we ultimately decided to bypass the execution of each thread entirely. While this is far from ideal, it proved to be the only practical solution within the time constraints of this study. The preferred approach would have been to repair and reintegrate the `PluginThread` mechanism. However, the plugin appears to have been developed for an earlier version of `SEMA`, making it difficult to understand and fix in a short timeframe.

Lost Execution Paths and Corrupted Returns

This issue encompasses two related problems. On one hand, during our analysis, `SEMA` frequently reported execution paths as "lost"—meaning symbolic execution reached a state where it could not proceed, often due to unresolved symbolic jumps or invalid instructions. On the other hand, we observed cases where the return value of certain functions became corrupted. As a result, when execution returned from these functions, it jumped to an unexpected or invalid location in the binary.

Both of these problems appear to be related to how return addresses or control flow information are managed within the symbolic execution engine. Unfortunately, diagnosing either issue is extremely challenging. In the case of corrupted returns, a careful inspection of execution traces can sometimes reveal the anomaly. However, for lost execution paths, no alert or diagnostic information is provided by `SEMA`, leaving analysts without visibility into the cause or location of the failure.

To address the issue of corrupted returns, we implemented several hooks to bypass problematic functions. However, the lost path issue remains largely unresolved, and

represents a serious limitation in terms of both debugging capability and overall path coverage.

Memory Limitations

This issue is likely related to the previously discussed problems, particularly the handling. Due to the extensive repetition , memory consumption became a progressively more serious concern as the analysis advanced. In the final stages of our experiments, the symbolic execution was often terminated not due to completion or imposed time constraint, but because it exhausted all available memory.

To mitigate this, we introduced several small patches and optimizations within the memory allocation call. These changes provided marginal improvements, and were insufficient to resolve the underlying issue. As a result, our analysis remains constrained by memory limitations, preventing further exploration of deeper paths.

SimProcedures

For this sample, we introduced 13 new `SimProcedures` and modified 15 existing ones.

Among the newly added procedures, 8 are dedicated to the management of the cryptographic context. These are summarized in the following table:

Table 4.14: Implemented SimProcedures for Cryptographic Functions

Call	**Implementation**
`CryptAcquireContextA`	Empty implementation
`CryptDecrypt`	Set key and size
`CryptDestroyKey`	Empty implementation
`CryptEncrypt`	Empty but return BVS
`CryptExportKey`	Set key and size
`CryptGenKey`	Set key
`CryptGetKeyParam`	Set key and size
`CryptImportKey`	Set key

In these `SimProcedures`, we consistently used a `BVS` as the cryptographic key with a fixed size. With the exception of `CryptEncrypt`—which, as marked in the table, returns a symbolic value of 0 and 1 to improve path coverage—all other procedures return a constant value of 1.

Ideally, for enhanced path exploration, each of these procedures would return a symbolic value like 0 and 1. However, due to the presence of loops tied to conditional logic, such a change would result in deeper loop execution, ultimately reducing overall path coverage in our context.

The remaining five newly implemented `SimProcedures` are listed in the following table:

Table 4.15: Additional Implemented SimProcedures

Call	Implementation	Return
`ExitThread`	Closes execution path	No return
`LocalFree`	Frees memory	0 on success, input on failure
`MoveFileW`	Empty	1
`SystemParametersInfoW`	Empty	BVS
`time`	BVS if output buffer	BVS

The following table presents the 15 `SimProcedures` that were modified during our analysis:

Table 4.16: Modified SimProcedures

Call	Modification
`??2@YAPAXI@Z`	Changed implementation and added commentary
`_local_unwind2`	Added part of the logic
`CreateFileA`	Fixed bugs in the logic
`CreateMutexA`	Modified return value
`CreateThread`	Special case, see associated paragraph above
`FindFirstFileW`	Changed commentary and modified a broken line
`free`	Small logic change, added error handling
`GetFileSize`	Adjusted return value for edge cases
`GetLastError`	Modified return value
`GetModuleFileNameA`	Fixed broken line
`GetModuleFileNameExA`	Fixed broken line
`GetTempFileNameA`	Fixed broken line and added symbolic verification
`GlobalAlloc`	Added maximum allocation size
`GlobalFree`	Implemented deallocation logic
`WriteFile`	Small comment change and added symbolic verification

4.3.3 Result Analysis

As shown previously, this binary contains approximately 388 unique external calls identified through static analysis. In the following table, we compare these results with those obtained through symbolic execution using `SEMA`.

As we can see, during our execution we were only able to recover approximately 43% of the unique external calls identified in the binary. This limitation primarily stems from the large number of loops within the code and the memory constraints encountered during symbolic execution.

Table 4.17: Estimated Number of External Calls per Phase (Static vs SEMA)

Phase	Static Calls	Unique Calls (SEMA)	Total Calls (SEMA)
Initialization	93	71 (+7)	124
Thread Creation	41	20	28
Encryption	254	76 (+3)	233
Total	**388**	**167 (+10)**	**385**

As before, these results do not take into account reinjected calls, and the numbers in parentheses represent obfuscated calls discovered by `SEMA`.

On another note, we observed an unusual inconsistency in our results. In four different cases, `SEMA` reported two distinct external calls at the same binary address. For example, at address `0x100027bc`, static analysis identified a call to `operator_delete` (`??3@YAXPAX@Z`), while `SEMA` resolved the same address as a call to both `operator_delete` and `SHGetFolderPathW`. This inconsistency may result from a bug in `SEMA`, issues within the involved `SimProcedure`s or hooks, or—less likely—self-modifying code. However, the most plausible explanation remains a bug in the symbolic analysis.

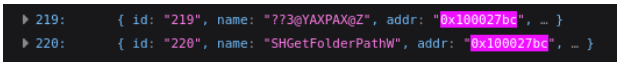

Figure 4.18: Example of Address Conflict: Static vs. Symbolic Analysis

4.4 Other sample

In this section, we study additional samples obtained from MalwareBazaar.

On MalwareBazaar, over 1,000 samples are tagged with the WannaCry signature. However, upon closer inspection, we found that many of these do not correspond to the exact WannaCry variant analyzed in this study. From a random selection of 20 samples labeled as WannaCry, 8 appeared to be instances of the `launcher.dll` component that we didn't analyze.

Among the remaining 12 samples, 3 were either empty or corrupted and provided almost no usable information during disassembly. One sample was completely different from known WannaCry binaries; it is unclear whether it was a heavily modified variant or a case of incorrect tagging. The remaining 8 samples appeared to match the WannaCry `mssecsvc.exe` component. We processed all 8 through SEMA using the same configuration as with our original sample to observe any behavioral differences.

In Table 4.19, Sample 0 corresponds to our original `mssecsvc.exe`. The remaining samples (1 through 8) were obtained from MalwareBazaar. Table 4.20 provides the SHA256 hash associated with each sample.

Table 4.19: Experimentation Results per Sample

Sample	Number of Calls	Execution Paths
0	68	8
1	62	6
2	62	6
3	62	6
4	88	12
5	66	8
6	62	6
7	62	6
8	62	6

Table 4.20: SHA256 Hash per Sample

Sample	SHA256
0	`24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c`
1	`2a62c57ba98308fe2316508f077186b76e5ad55a1e367e58d19e5a9b08900eec`
2	`3f48a8d80cc55a1fbe9a210b60b07f3677b736b8a02d5408697d9df54a276776`
3	`51d3aa054c3c98e25e973f16a75b267b1b4823cb5edd9ba0fedd85f12a44567c`
4	`427c05a71e319854d7e14101eae7bf57f08301b232fd113b3a9e0ce00ac9e9a9`
5	`b48f682b2eb280061211435817f9a0f74431eb6a3841bb506b0614be524e7fdb`
6	`cb11ca7f0afe82833d91792dc891a81088f1605c6cc029edecc21b4f433c0756`
7	`ce2194c96ebab334f8484a7a3e45e2c3bb74296fc5eddd335abf3f5c65f34967`
8	`e06afc45a77a51ff9c8ab94fcd5a4777af1cc374e2d9f73b91a1780bfa42e3fe`

As we can see, 6 out of the 8 new samples (Samples 1–3 and 6–8) produce exactly the same number of execution paths and external calls. These appear to be near-identical variants of the original binary, with only minor differences. One such difference is that these variants bypass the kill switch. While our original sample contains a clear kill switch check at the start of execution (Figure 4.1), these modified samples implement a slightly altered version, as shown in Figure 4.21.

This change disables the conditional branch responsible for halting execution, which in turn reduces the number of execution paths and repeated external calls observed by SEMA.

Sample 4 is structurally the same as the others but produces significantly more execution paths (12 compared to 6 in others) and more external calls (88). The increased execution path count may result from minor differences in variables or internal logic that influence symbolic branching.

```
local_1 = 0;
uVar1 = InternetOpenA(0,1);
InternetOpenUrlA(uVar1,&uStack_64,0,0,0x84000000,0);
InternetCloseHandle(uVar1);
InternetCloseHandle(0);
FUN_00408090();
```

Figure 4.21: Modified kill switch implementation in certain variants

Sample 5 appears to be identical to Sample 0 but exhibits two fewer external calls. The reason for this discrepancy is unclear; it may stem from subtle differences between the two samples that are not immediately apparent through standard analysis.

Normally, for a given sample, the number of execution paths and external calls should be fixed and consistent. As observed, Samples 1 to 3 and 5 to 8 appear to be perfectly identical. In contrast, the slight differences between Samples 4 and 5 may be due to minor variations in the binaries, which could explain the discrepancies in the results.

However, during the analysis, we discovered that some external calls listed by SEMA are repetitive and should not appear in the final output. This indicates a minor issue in SEMA's filtering mechanism. While this does not pose a critical problem in its current form, it can hinder precise analysis and interpretation of the results.

4.5 Conclusion

This chapter presented the analysis of the `WannaCry` ransomware using the `SEMA` Toolchain. Due to the multi-layered structure of the ransomware, we analyzed three distinct components, each introducing its own logic and challenges.

The first component, `mssecsvc.exe`, did not present any major blocking issues. We were able to achieve full path coverage for this component (excluding its worm functionality). This analysis also revealed certain limitations in `SEMA`, most notably the inability to use the resource plugin, which led to the corruption of the `tasksche.exe` created during the execution. On the positive side, we successfully identified the entry point of the worm component using both static and symbolic analysis, although we did not analyze it in depth due to its being outside of scope of this study.

In contrast, we were unable to achieve full path coverage for the second (`tasksche.exe`) and third (`kbdlv`) components. In the case of `kbdlv`, the primary obstacle was the large number of loops, which significantly hindered symbolic execution. For `tasksche.exe`, the difficulties were more complex and stemmed from a combination of embedded C++ logic and `SEMA`'s limitations in resolving complex pointer operations.

Additionally, we encountered several issues related to the VEX engine, which caused crashes at the end of some functions. These crashes were difficult to diagnose, as the root causes could not be clearly identified.

To improve future analyzes, we propose several enhancements to `SEMA`. First, fixing the resource plugin would prevent corruption of created file during analysis. Second, implementing a plugin to facilitate loop skipping would improve scalability and execution

depth. Furthermore, incorporating concolic execution could provide an effective workaround for both VEX-related crashes and loop-heavy code. By switching to concrete execution in problematic scenarios, such as complex pointer logic within loops, it would be possible to accelerate execution and avoid infinite symbolic loops.

This concludes our case study of `WannaCry` and sets the stage for the broader conclusions discussed in the next chapter.

Chapter 5

Conclusion

This work explored the use of symbolic execution to analyze ransomware behavior using the `SEMA` toolchain. Our primary objective was to assess the capabilities and limitations of `SEMA` when applied to two real-world malware samples: `GonnaCry` and `WannaCry`.

As part of this effort, we integrated existing research into the main version of `SEMA`. Additionally, we developed and implemented a new plugin to address a key limitation introduced by the use of function hooks. Specifically, this plugin re-inserts missed external calls after symbolic execution, mitigating the information loss that results from function interception.

The first case study, `GonnaCry`, provided a valuable opportunity to gain practical experience with the toolchain and deepen our understanding of symbolic execution. We successfully achieved full path coverage, demonstrating `SEMA`'s capability to fully analyze a ransomware sample under controlled conditions.

The second case study, `WannaCry`, presented greater complexity and highlighted both the strengths and limitations of symbolic execution. On the one hand, we achieved full path coverage for the `mssecsvc.exe` component (excluding worm functionality), showcasing `SEMA`'s ability to reveal obfuscated or hidden calls that static analysis might miss. On the other hand, we were unable to achieve full path coverage for the `tasksche.exe` and `kbdlv` components. These challenges underscored limitations in both the toolchain and our implementation.

Throughout our analysis, we identified several issues and areas for improvement in `SEMA`. We believe the toolchain would greatly benefit from the integration of concolic execution, which could mitigate some of the difficulties encountered with complex loops and VEX-related crashes. Additionally, the introduction of a plugin that allows selective skipping of code regions—without the need for cumbersome function hooks—would enhance usability and analysis efficiency.

Moreover, although not discussed in detail within the analysis chapters, we encountered recurring issues with plugin and `SimProcedure` compatibility. Many components developed for earlier versions of `SEMA`, such as the resource plugin or the thread plugin, were no longer functional due to version mismatches and a lack of ongoing maintenance. This suggests that a significant refactoring of the toolchain is necessary to improve modularity, ensure

long-term maintainability, and support future extensions.

In conclusion, this work demonstrated the potential of symbolic execution through `SEMA` in the context of ransomware analysis, while also identifying key areas for improvement that could enhance its reliability, flexibility, and effectiveness in real-world malware research.

Bibliography

[1] European Union Agency for Cybersecurity (ENISA). *ENISA Threat Landscape 2024*. accessed on June 1st, 2025. 2024. URL: https://www.enisa.europa.eu/sites/default/files/2024-11/ENISA%20Threat%20Landscape%202024_0.pdf.

[2] BlackFog. *The State of Ransomware - January 2025*. https://www.blackfog.com/the-state-of-ransomware-2025/#January. accessed on June 1st, 2025.

[3] Harun Oz et al. "A survey on ransomware: Evolution, taxonomy, and defense solutions". In: *ACM Computing Surveys (CSUR)* 54.11s (2022), pp. 1–37.

[4] Charles-Henry Bertrand Van Ouytsel et al. "Tool Paper - SEMA: Symbolic Execution Toolchain for Malware Analysis". In: *Risks and Security of Internet and Systems*. Springer Nature Switzerland, 2023, pp. 62–68. ISBN: 9783031311086. DOI: 10.1007/978-3-031-31108-6_5. URL: http://dx.doi.org/10.1007/978-3-031-31108-6_5.

[5] Fabrizio Biondi et al. "Tutorial: An Overview of Malware Detection and Evasion Techniques". In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Springer International Publishing, 2018, pp. 565–586. ISBN: 9783030034184. DOI: 10.1007/978-3-030-03418-4_34. URL: http://dx.doi.org/10.1007/978-3-030-03418-4_34.

[6] Serena Lucca and Axel Legay. "Let's analyze malware with symbolic execution: a practical study". PhD thesis. uclouvain, 2022. URL: https://dial.uclouvain.be/downloader/downloader.php?pid=thesis%3A36890&datastream=PDF_01&cover=cover-mem.

[7] Serena Lucca et al. "On Exploiting Symbolic Execution to Improve the Analysis of RAT Samples with angr". In: *Foundations and Practice of Security*. Springer Nature Switzerland, 2024, pp. 339–354. ISBN: 9783031575372. DOI: 10.1007/978-3-031-57537-2_21. URL: http://dx.doi.org/10.1007/978-3-031-57537-2_21.

[8] Fabio Gritti et al. "SYMBION: Interleaving Symbolic with Concrete Execution". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, June 2020. DOI: 10.1109/cns48642.2020.9162164. URL: http://dx.doi.org/10.1109/CNS48642.2020.9162164.

[9] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. URL: http://dx.doi.org/10.1145/1095430.1081750.

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

[11] Lockshaw Subwire. *throwing a tantrum, part 1: angr internals*. https://angr.io/blog/throwing_a_tantrum_part_1/. accessed on June 1st, 2025.

[12] Craig Beaman et al. "Ransomware: Recent advances, analysis, challenges and future research directions". In: *Computers & security* 111 (2021), p. 102490.

[13] Helen Jose Chittooparambil et al. "A review of ransomware families and detection methods". In: *Recent Trends in Data Science and Soft Computing: Proceedings of the 3rd International Conference of Reliable Information and Communication Technology (IRICT 2018)*. Springer. 2019, pp. 588–597.

[14] Philip O'Kane, Sakir Sezer, and Domhnall Carlin. "Evolution of ransomware". In: *Iet Networks* 7.5 (2018), pp. 321–327.

[15] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[16] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[17] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 263–272. ISBN: 1595930140. DOI: 10.1145/1081706.1081750. URL: https://doi.org/10.1145/1081706.1081750.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[19] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

[20] Nick Stephens et al. "Driller: Augmenting fuzzing through selective symbolic execution". In: *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “The S2E platform”. en. In: *ACM Trans. Comput. Syst.* 30.1 (Feb. 2012), pp. 1–49.

[22] Jiri Slaby, Jan Strejček, and Marek Trtík. “Symbiotic: synergy of instrumentation, slicing, and symbolic execution”. In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'13. Rome, Italy: Springer-Verlag, 2013, pp. 630–632. ISBN: 9783642367410. DOI: 10.1007/978-3-642-36742-7_50. URL: https://doi.org/10.1007/978-3-642-36742-7_50.

[23] Roberto Baldoni et al. “Assisting Malware Analysis with Symbolic Execution: A Case Study”. In: *International Conference on Cyber Security Cryptography and Machine Learning*. 2017. URL: https://api.semanticscholar.org/CorpusID:41540186.

[24] Serena Lucca et al. “On exploiting symbolic execution to improve the analysis of rat samples with angr”. In: *International Symposium on Foundations and Practice of Security*. Springer. 2023, pp. 339–354.

[25] *EVP_EncryptInit - OpenSSL Documentation*. https://docs.openssl.org/1.0.2/man3/EVP_EncryptInit. accessed on June 1st, 2025.

[26] Cloudflare. *What is WannaCry ransomware?* accessed on June 1st, 2025. n.d. URL: https://www.cloudflare.com/learning/security/ransomware/wannacry-ransomware/.

[27] Marcus Hutchins. *How to Accidentally Stop a Global Cyber Attacks*. accessed on June 1st, 2025. 2017. URL: https://www.malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html.

[28] Brandefense. *Lazarus APT Group (APT38)*. accessed on June 1st, 2025. n.d. URL: https://brandefense.io/blog/apt-groups/lazarus-apt-group-apt38/.

[29] The Independent. *NHS ‘could have prevented’ WannaCry malware hack*. accessed on June 1st, 2025. 2017. URL: https://www.independent.co.uk/news/uk/home-news/wannacry-malware-hack-nhs-report-cybercrime-north-korea-uk-ben-wallace-a8022491.html.

[30] Check Point Research. *Brokers of the Shadows: Analyzing the Vulnerabilities and Attacks Spawned by Leaked NSA Hacking Tools*. accessed on June 1st, 2025. 2017. URL: https://research.checkpoint.com/2017/brokers-shadows-analyzing-vulnerabilities-attacks-spawned-leaked-nsa-hacking-tools/.

[31] Shou-Ching Hsiao and Da-Yu Kao. “The static analysis of WannaCry ransomware”. In: *2018 20th International Conference on Advanced Communication Technology (ICACT)*. 2018, pp. 153–158. DOI: 10.23919/ICACT.2018.8323680.

[32] stacksmashing. *Reversing WannaCry Part 2 - Diving into the malware with #Ghidra*. accessed on June 1st, 2025. 2020. URL: https://youtu.be/Q90uZS3taG0.

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