



School of Engineering

InIT Institute of Applied
Information Technology

Bachelor thesis (Computer Science)

Ghidrion: A Ghidra Plugin to Support Symbolic Execution

Author

Silvan Flum
Valentin Huber

Main supervisor

Arno Wagner

Sub supervisor

Gürkan Gür

Industrial partner

Cyber-Defence (CYD) Campus Zürich, armasuisse S+T

External supervisor

Damian Pfammatter

Date

09.06.2023

DECLARATION OF ORIGINALITY

Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:

Zürich, 09.06.2023

Zürich, 09.06.2023

Name Student:

Silvan Flum

Valentin Huber

Abstract

Symbolic execution is a powerful technique for automatic analysis of and reasoning about program behaviour, particularly in binary analysis. However, popular reverse engineering tools such as Ghidra lack native support for symbolic execution. Existing extensions advertising symbolic execution are limited in functionality, do not scale well enough to be employed on practical binaries and provide limited documentation. The Cyber-Defence Campus of armasuisse, as part of its vulnerability research program, has developed a proof-of-concept tool called Morion, that enables symbolic execution-based analysis of various vulnerability types on practical binaries. Previously, it had to be configured by manually writing configuration files. This thesis proposes Ghidrion, an open-source Ghidra plugin that leverages information gathered from Ghidra's analysis tools to enhance analysts' usage of Morion. Ghidrion suggests calls to external functions that can be hooked and simplifies configuring the setup necessary to run Morion. It further supports the analysis of Morion's results by visually highlighting executed instructions and providing a side-by-side comparison of memory and register values at the beginning and end of the execution. Alongside the code, previously missing documentation on developing Ghidra plugins is provided. This thesis further proposes future research directions, such as improvements to Ghidra's loader to match external functions to their libraries and added support for interactive Python shells to run Morion's analysis modules from within Ghidra.

Keywords: Symbolic Execution, Ghidra, Vulnerability Research, Binary Analysis, Software Reverse Engineering

Foreword

Writing this thesis has been a great introduction to software reverse engineering and binary analysis. The knowledge and insights gained will be foundational to our careers, and we are grateful to have gotten this opportunity.

We would like to thank Damian Pfammatter from the Cyber-Defence Campus and Arno Wagner from Zurich University of Applied Sciences for their support and understanding throughout the different phases of this project, Alexandra Gunz and Frawa Vetterli for their comments on early versions of the thesis, and Gürkan Gür for his support.

Contents

1	Introduction	1
1.1	Motivation and Project Background	1
1.2	Fundamentals	1
1.3	Example Problem	2
1.4	Thesis Outline	3
2	Theoretical Principles	4
2.1	Software Reverse Engineering	4
2.2	Binary Analysis	5
2.2.1	Static Binary Analysis	5
2.2.2	Dynamic Binary Analysis	5
2.2.3	Hybrid Binary Analysis	6
2.3	Symbolic Execution	6
2.3.1	From Concrete to Symbolic Execution	6
2.3.2	Symbolic Execution Example	7
2.3.3	Variants of Symbolic Execution	9
2.3.4	Symbolic Execution and Taint Analysis	11
3	State of the Art	12
3.1	Introduction to Symbolic Execution Engines	12
3.1.1	angr	12
3.1.2	Triton	13
3.1.3	Conclusion	14
3.2	Symbolic Execution Graphical User Interfaces	14
3.2.1	AngryGhidra	14
3.2.2	angr Management	16
3.2.3	Ponce	17
3.2.4	Manticore User Interface	18
4	Concept and Approach	21
4.1	Morion	21
4.1.1	Workflow	21
4.1.2	Interface	24
4.2	Parts of the Plugin	25
4.2.1	Creating an Init YAML File	25
4.2.2	Analysing a Traced YAML File	25
5	Architecture and Implementation	27
5.1	Extending Ghidra	27
5.1.1	GhidraDev	27
5.1.2	Ghidra Scripts	27

5.1.3	Ghidra Module Projects	28
5.1.4	Ghidra's Program API	29
5.1.5	Context Menus in Ghidra	30
5.2	Architecture and Technology	30
5.2.1	Java/Swing	31
5.2.2	Model-View-Controller Pattern	31
5.2.3	Observer Pattern	32
5.2.4	Graphical User Interface Design	32
5.2.5	Python Integration	32
6	Results	33
6.1	Created Documentation	33
6.2	Creating an Init YAML File	33
6.2.1	Adding Hooks	33
6.2.2	Adding Memory	34
6.2.3	Adding Registers	36
6.2.4	Saving and Importing Init Trace Files	36
6.3	Tracing	38
6.4	Analysing a Trace	38
6.4.1	Differences Between Entry and Leave States	38
6.4.2	Marking Visited Instructions	38
6.4.3	Using Morion's Analysis Modules	38
7	Discussion and Outlook	41
7.1	Created Documentation	41
7.2	Improvements in the Workflow of an Analyst	41
7.2.1	Create Init YAML	41
7.2.2	Tracing	42
7.2.3	Analyse Traced YAML	42
7.3	Future Work	42
7.3.1	Hooking Functions in Sections Other Than <code>.text</code>	42
7.3.2	Automatic Library Detection	42
7.3.3	Automatic Register Detection	43
7.3.4	Further Ideas	44
Indices		45
1	Bibliography	45
2	List of Figures	50
3	List of Tables	50
4	List of Listings	50
Appendix		51
1	Installation of AngryGhidra	51
2	Full Example Traced YAML File	52
3	Full Code Proposed to Load External Symbols	55
4	Registers Identified by Ghidra	56
5	Initial Thesis Description	61
5.1	Titel	61
5.2	Beschreibung	61
5.3	Voraussetzungen	62

Chapter 1

Introduction

This chapter first provides the project background. It then explains the fundamentals of the relationship between human-readable and machine-executable code and gives an overview of binary analysis and symbolic execution. Third, it introduces an example problem that is used throughout this thesis. Finally, it gives an overview of the rest of this thesis.

1.1 Motivation and Project Background

The Cyber-Defence (CYD) Campus [1] was tasked with checking binaries of Internet of Things (IoT) devices for vulnerabilities. As part of this research, they developed a series of Python scripts to automatically analyse these binaries using symbolic execution. As these scripts accumulated, they were eventually merged into a proof-of-concept tool called Morion. Although Morion is still in its infancy, it can already be used to examine binaries.

So far, however, using Morion has been rather cumbersome and laborious. The analysis setup had to be done by manually writing configuration files. For this reason, the CYD Campus commissioned the authors of this thesis to create a plugin for Ghidra [2] that simplifies the use of Morion. Ghidra is a popular reverse engineering tool with a user interface that provides a disassembler and decompiler among many other features. Because of this, analysts often already employ it alongside Morion. Ghidra further supports the development of extensions that allow developers to programmatically access the information obtained by Ghidra's analysis modules.

1.2 Fundamentals

Most programmers develop computer programs using high-level languages such as C, Java, or Python [3]. And while these high-level languages are intuitive to humans, computers do not understand them. Computers only understand machine language, which is just binary data, a series of zeros and ones. It is almost impossible for humans to understand machine language. Another problem is that machine language differs from processor architecture to processor architecture. [4] As a result, a program written in machine language for a processor with architecture X will not be understood by a processor with a different architecture Y. To solve this problem, a translator is needed.

An assembler is a program that translates assembly language into machine language. Assembly is designed to be human-readable and consists of commands that can each be directly translated to one machine language instruction. Although code written in assembly is less cryptic than machine code because it uses names for different instructions instead of just numbers, it is still far from intuitive. In addition, assembly language is still architecture specific. [4]

Compilers are programs that solve both problems by translating code written in a high-level programming language into binary machine code. Programs developed in a high-level language only need to be written once, because a compiler can usually translate the code into machine code for different architectures. [4] The files created by the compiler containing the machine code are called binary executables, or binaries for short [3]. These can then be executed by a computer.

To reverse these steps and go from binary to human-readable code, so-called disassemblers and decompilers are employed. However, interpreting the results of these still is a hard problem. First, programs sometimes contain thousands of files, each of which may contain hundreds of lines of code. Second, the logic of a program is not always easy to understand. In particular, decompilers often produce code that is not easily readable. As a result, binary analysis techniques have been developed to allow the automatic examination of binaries. One such technique is called symbolic execution, or "symbex" for short.

Symbolic execution allows an analyst to automatically answer complex questions about the behaviour of a program [3] because often a lot of inputs have an equivalent effect and can therefore be thought of as a class of inputs. The analysis then only has to be done once for each class. Symbolic execution is a complex technique and not easy to implement. Therefore, the binary analysis community has made efforts to develop so-called symbolic execution engines. These engines often provide symbolic execution capabilities for a variety of computer architectures.

1.3 Example Problem

The example code provided in Listing 1.1 is used throughout this thesis for illustrative purposes. The buffer allocated in line 15 is (partially) marked as symbolic, allowing an analyst to calculate what values it needs to be set to in order to change the result of the comparison in line 18. In the context of vulnerability research, the buffer can be thought of as the equivalent of chunks of memory in real binaries that can be controlled by an attacker's input. The comparison in line 18 would then be a control that they would like to bypass, such as a password check.


```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BUF_LENGTH 8
6
7 /*
8  * size_t strlen(const char *s);
9  */
10
11 int main(int argc, char *argv[]) {
12     char *s;
13
14     // Buffer, will be (partially) symbolised
15     s = (char *) calloc(BUF_LENGTH, sizeof(char));
16
17     // Testing strlen
18     if(strlen(s) == 2) {
19         printf("strlen('%s') == 2\n", s);
20         return EXIT_SUCCESS;
21     }
22     printf("strlen('%s') != 2\n", s);
23     return EXIT_SUCCESS;
24 }
```

LISTING 1.1: Example Code — `strlen.c`

1.4 Thesis Outline

Following the introduction, this thesis is organized into six main sections.

- Chapter 2 introduces the theoretical foundation for the rest of the thesis. It explains what software reverse engineering and binary analysis are and why one may want to use them. It continues with a section on symbolic execution, its use cases, and different variants and approaches.
- Chapter 3 consists of a survey of existing symbolic execution engines and graphical user interfaces (GUIs) for reverse engineering tools that support symbolic execution.
- Chapter 4 introduces Morion, including a detailed explanation of its current workflow and interface. This chapter further describes how Ghidion, the Ghidra plugin developed as part of this thesis, interacts with Morion.
- Chapter 5 deals with the technical aspects of extending Ghidra using its Application Programming Interface (API). It further outlines the architecture implemented and design principles used in the development of Ghidion.
- Chapter 6 demonstrates the capabilities of Ghidion, shows the achieved features in detail, and discusses the improved interface supporting the use of Morion.
- Chapter 7 reviews the contribution of this thesis and discusses the improvements made by Ghidion. Finally, the authors discuss ideas that could further enhance the interaction between Morion and Ghidra.

Chapter 2

Theoretical Principles

The sections of this chapter provide the theoretical foundation upon which the implementation and results of this thesis are based. The relevant concepts and theories are introduced from top to bottom, starting with software reverse engineering, through binary analysis, to the different variants of symbolic execution.

2.1 Software Reverse Engineering

Software Reverse Engineering (SRE) is the practice of extracting design and implementation information from a (part of a) software system [5]. It can alternatively be described as the process of identifying the components of a software system and their dependencies and creating a different form of representation, for example at a higher level of abstraction [6]. There are two main categories in which software reverse engineering is used [5].

First, software engineers use it to understand software when they need to maintain or develop it [6]. Eilam [7] identified four scenarios, explained in detail by Cipresso and Stamp [5], in which a software developer may use reverse engineering:

- Developing applications that use proprietary software.
- Reversing developed software into an abstract design to verify that the implementation conforms to the original design.
- Performing binary analysis to evaluate software quality and robustness.
- Recovering unavailable source code for software maintenance and development.

Second, it has applications in software security. Antivirus developers use reverse engineering to identify and understand viruses and malware [5]. Eilam [7] lists four scenarios, again explained in detail by Cipresso and Stamp [5], in which reverse engineering is used in a software security environment:

- Detecting and neutralizing viruses and malware.
- Testing cryptographic algorithms for weaknesses.
- Protecting proprietary software and digital rights with anti-reversal techniques.

- Testing and analyzing the security of program binaries.

The last of the above, testing and analyzing the security of program binaries, is one application of binary analysis. In the following section, binary analysis is described in more detail, since symbolic execution is a technique that builds on binary analysis. And as mentioned in Section 1.2, Morion itself builds on symbolic execution.

2.2 Binary Analysis

Andriessse [3] defines binary analysis as "the science and art of analyzing the properties of binary computer programs and the machine code and data they contain". In the context of this definition, it is important to note that binary analysis, software reverse engineering, and disassembly are not synonymous. Although they are related, and often used together, and there is not always an agreement in the literature about the relationship between the three terms, they are by no means the same. [3] Software reverse engineering does not have to be at the binary level [5]. And if so, it is only one application of binary analysis. Others include debugging, performance analysis, software reliability, digital forensics, and security analysis. [8] Disassembly, on the other hand, is usually the first step in binary analysis [3]. However, it only transforms data, no actual analysis takes place yet.

After all, the goal of binary analysis is to find out what a binary does when executed [3]. Since compilers may contain bugs, even the availability of source code does not negate the necessity of binary analysis in certain contexts. Therefore, the semantics of an executed binary may differ from the expected semantics of the source code. [8] There are two complementary approaches to analyzing binaries, static analysis and dynamic analysis [9].

2.2.1 Static Binary Analysis

Static binary analysis only analyzes the machine code of a binary without executing it [9]. The advantages are that one can analyze all the execution paths of a binary without having to run it repeatedly. In addition, static analysis is architecture independent. This means, for example, that an ARM binary can be analyzed on an x86 machine. These advantages come with a series of disadvantages. First, there is no runtime information available. [3] This means that possible values and their types have to be derived from the machine code. This is a hard task because the compilation process discards much of the information present in the source code. Second, on top of the binary itself, additional complexity such as dynamic linking or specialized assembly instructions must also be taken into account. [8]

2.2.2 Dynamic Binary Analysis

Dynamic binary analysis executes the binary and analyzes it on the fly [3]. This is done by adding extra analysis code to the binary [9]. Often this is done by hooking, where function calls are intercepted and then corresponding hook functions are also called. These hook functions contain the analysis code. [8] Apart from slowing down the execution, the analysis code does not interfere with the execution [9]. Although this process may sound complex, dynamic analysis is simpler and more accurate than static analysis [3], [9]. The reason for this is that, unlike static analysis, the entire runtime state is known. The downside is that only one path is analyzed per

execution, and all other possible paths are unknown. So it is possible that interesting paths will be ignored. [3]

2.2.3 Hybrid Binary Analysis

There is also a combination of these two techniques called hybrid analysis. After the binary is statically analyzed it is executed and dynamically analyzed. This provides advantages because the possible execution paths are known from the static analysis and can be selectively traversed during the dynamic analysis. This process can also be repeated several times if, for example, the dynamic analysis provides new findings. [8]

2.3 Symbolic Execution

Symbolic execution, or symbex for short, is a powerful and popular software analysis technique [3]. It is used in automated software testing to generate tests and in software, security to analyze a program's behavior [10]. Symbolic execution can automatically create test inputs that explore new execution paths and thus increase code coverage [3]. By exploring new execution paths, bugs and unintended behaviours can be identified, which in turn can lead to the identification of exploitable vulnerabilities. For example, symbolic execution can verify that software never performs a division by zero, or that it is impossible to bypass authentication. [11] And if an unintended behaviour is detected, it can be easily reproduced thanks to the metadata collected during symbolic execution [10]. In conclusion, symbolic execution has its place in software security. This section reviews the basics of symbolic execution, provides a simple example, and discusses its variations, challenges, and limitations.

2.3.1 From Concrete to Symbolic Execution

Normally, a program is executed on concrete values (stored in registers and memory locations in the context of binaries) such as an integer 23 or a string "ghidrion" [3]. In doing so, it explores exactly one path per execution [11]. In contrast, when a program is executed symbolically, some or all variables are symbolized, meaning they are represented by a symbol α_i . Such a symbol represents every possible value that a variable (register or memory location in the context of binaries) can take at a time and place. During symbolic execution of a program, the mathematical operations performed on a symbol are recorded as logical formulas. Taken together, these symbols and logical formulas are the collected metadata needed to form what is called the symbolic state. [3]

Computing the Symbolic State

The symbolic state consists of a set of symbolic expressions, called a symbolic expression store σ , and a path constraint π . A symbolic expression ϕ_j is either a symbol α_i or an arithmetic combination of multiple symbolic expressions, such as $\phi_j = \phi_k - \phi_l$, where $i, j, k, l \in \mathbb{N}$. The symbolic state also records the mappings of variables to symbolic expressions. The path constraint, on the other hand, is the conjunction of all branch constraints. A branch constraint represents the relational and logical operations performed on symbolic expressions in a single branch. [3] For example, Listing 2.1 contains the two **if** statements **if**(len1 >= 2) and **if**(len2 <= 5). So, the

corresponding two branch constraints are $\phi_1 \geq 2$ and $\phi_2 \leq 5$, where both conditions evaluate to `true`, ϕ_1 maps to `len1`, and ϕ_2 maps to `len2`. Consequently, the path constraint to reach line 5 becomes $\phi_1 \geq 2 \wedge \phi_2 \leq 5$.

```

1  int len1 = strlen(argv[1]);
2  int len2 = strlen(argv[2]);
3  if (len1 >= 2) {
4      if (len2 <= 5) {
5          between();
6      }
7      if (len1 < 2) {
8          impossible();
9      }
10     greater();
11 }
12 less();

```

LISTING 2.1: Pseudocode Including a Nested `if` Statement

To solidify the understanding, the next section will demonstrate a symbolic execution.

2.3.2 Symbolic Execution Example

For simplicity, this example is performed using the pseudocode of Listing 2.1 and only symbolizes the variables `len1` and `len2`. The argument vector `argv` is concrete. Note that Morion and the tools presented in the next chapter operate with binaries instead of high-level source code. Figure 2.1 shows the evolution of the symbolic state for each line of code.

Initially, there is no path constraint π and the symbolic expression store σ is empty. By assigning `len1`, there is still no path constraint, but `len1` is made symbolic and stored as a symbolic expression ϕ_1 with symbolic value α_1 in the symbolic expression store. Likewise, the mapping from `len1` to ϕ_1 is stored. The same happens with the assignment of `len2`. It gets interesting again after the first `if` statement, where a new symbolic state is created for each path and the path constraint is updated according to the `if` condition. This process continues until the entire pseudocode has been run through. After that, constraint solving can be applied to determine if a statement is reachable.

Constraint Solving

Suppose the goal of the analyst is to call the function `between()` on line 5. The symbolic state of `between()` in Figure 2.1 shows that the path constraint $\pi := \phi_1 \geq 2 \wedge \phi_2 \leq 5$ must be solved. In addition, the symbolic expressions ϕ_1 and ϕ_2 are symbolic values (and not a combination of other symbolic expressions) and represent `len1` and `len2` respectively. For example, a possible solution is $\alpha_1 = 2 \wedge \alpha_2 = 0$. Such a solution is called a model. [3] When the pseudocode is executed concretely, `between()` is reached by the assignments `len1 = 2` and `len2 = 0`.

This example is kept very simple and the constraints can be solved without the help of a computer since only two `if` statements were encountered, each with a simple

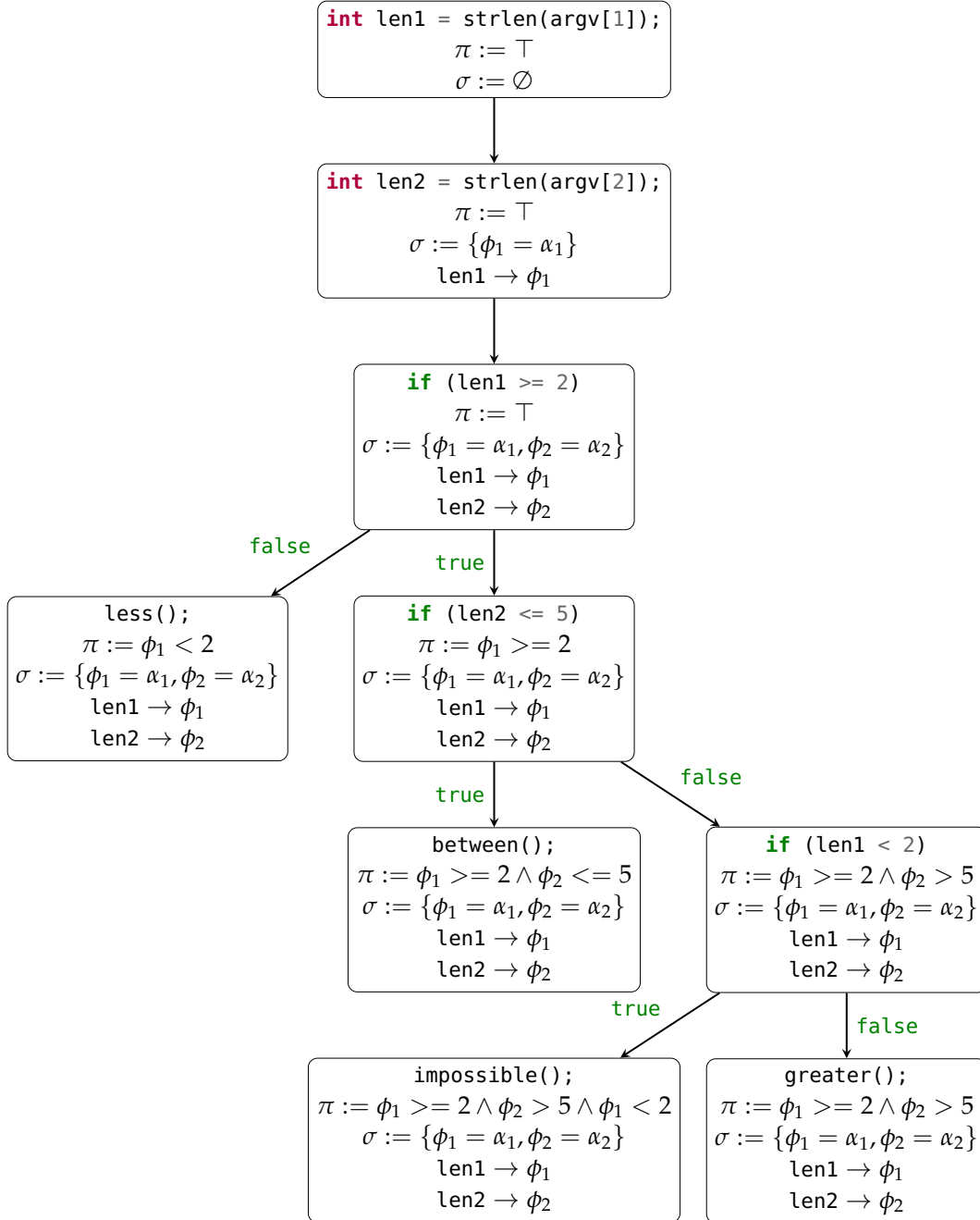


FIGURE 2.1: An Illustration Inspired by Andriesse [3] Showing the Evolution of the Symbolic State for All Execution Paths in Listing 2.1.

relational condition. However, in a real-world program execution path, there are often thousands of control structures with more complex conditions traversed. Solving these constraints is no longer feasible without the help of computers. Therefore, this work is left to so-called satisfiability modulo theories (SMT) solvers such as Z3 [12], also known as constraint solvers.

2.3.3 Variants of Symbolic Execution

The example above was done statically because we did not compile and run the pseudocode. This is the traditional way of doing symbolic execution [3]. However, symbolic execution can be done in a number of different ways. According to Andriessse [3], the four most important dimensions of how to perform symbolic execution are:

- Static vs. dynamic
- Online (parallel) vs. offline (non-parallel)
- Symbolic state
- Path coverage

Figure 2.2 shows in which combinations the different dimensions can be used.

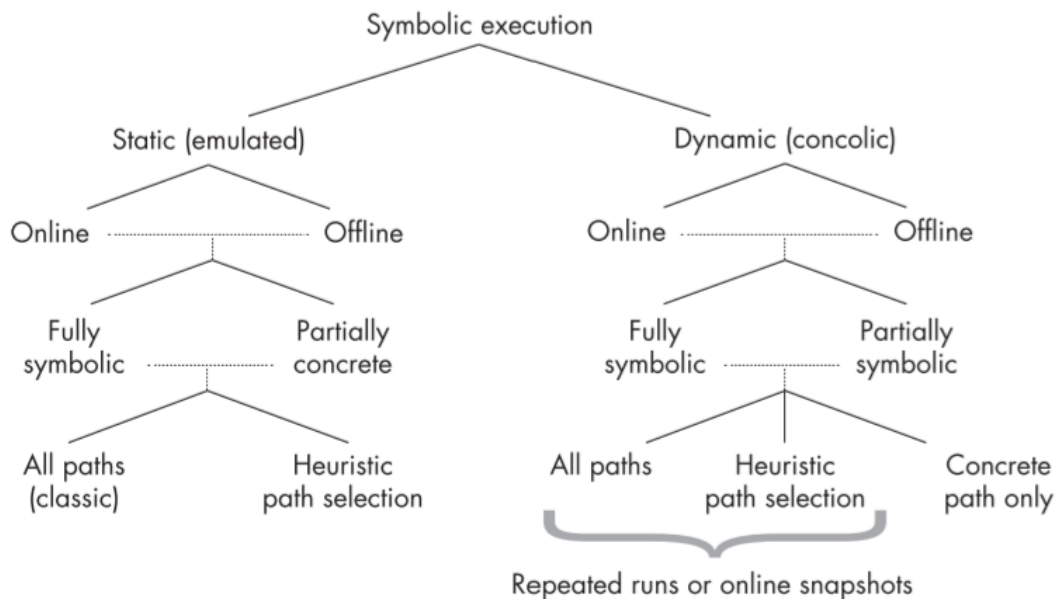


FIGURE 2.2: The Design Dimensions in Symbolic Execution According to Andriessse [3]

Static Symbolic Execution (SSE)

As with binary analysis, discussed in 2.2, symbolic execution can be performed statically or dynamically. Static Symbolic Execution is the original form of symbolic execution and exhaustively explores all possible paths of a program or uses heuristics to analyze only a subset of all paths. SSE is usually online, meaning that multiple paths are explored in parallel. [3] It is worth noting that, for example, if an analyst is trying

to reach a specific location in a program, an exhaustive symbolic execution finds all inputs that lead to that location. In addition, no inputs are reported as possible that do not even reach that location. In other words, false negatives and false positives are prevented. [11]

An advantage of this variant is that programs of an architecture X can be executed symbolically on a machine with a different architecture Y. Another advantage is the ability to analyze a single component of a program. [3] On the downside, SSE has two significant problems. First, the number of execution paths grows exponentially with the number of conditionals. As a result, the maintained symbolic state grows at the same rate. [10] These effects, called path explosion [10] and state space explosion [11], respectively, lead to scalability issues. Heuristics to decide which paths to explore can help in this respect, but only to a limited extent, because determining reasonable and efficient heuristics is a complex task in itself. Second, when the symbolic execution flows to software components that are not under control, such as the kernel or shared libraries, the behaviour of these components must be modelled. [3] However, such modelling is not only difficult but also time-consuming since the interfaces to libraries and the OS can be very extensive [10].

Dynamic Symbolic Execution (DSE)

Dynamic symbolic execution is a combination of concrete and symbolic execution. For this reason, DSE is also called concolic ("concrete symbolic") execution. This variant runs a program with concrete values but additionally computes the symbolic state in parallel. As a result, unlike SSE, DSE can only analyze one path at a time. After exploring a path, DSE takes the path constraint and inverts one of its branch constraints. A constraint solver then computes new concrete values over the changed path constraint. By running the program with these new concrete values, a new path will be explored. [3]

The two problems with static symbolic execution are improved or even solved by dynamic symbolic execution. First, since a concrete execution only traverses one path at a time, DSE is usually offline. This does not prevent path explosion, since in the end all paths still have to be traversed. However, it does prevent state space explosion, because only the symbolic state of exactly one path needs to be computed at a time. Therefore, DSE scales better than SSE. Second, software components that are not under control do not have to be modelled, but can simply be executed concretely. The major drawbacks are that code coverage may depend on the concrete values chosen and that the program can only be executed on the architecture for which it was compiled. [3]

Online vs. Offline Symbolic Execution

As mentioned above, SSE is typically online and DSE is offline. Online symbex explores multiple execution paths in parallel, while offline symbex explores only one at a time. Since all paths must be analyzed either way, neither option can prevent a path explosion. However, the case is different for the state space explosion. State space explosion results directly from the parallel (online) path exploration and thus does not occur in offline symbex. There is an advantage to the online symbex, however. Online symbex executes each program instruction only once. On the contrary, offline symbex usually has to execute many instructions multiple times, because the program has to be executed from the beginning for each path. [3]

Symbolic State

In principle, it is possible to represent each register and memory address symbolically. However, this quickly leads to constraints that are difficult to solve. Therefore, often only some parts (registers and memory locations in the context of binaries) of a program are marked as symbolic. Consequently, the analyst has to decide which parts to symbolize. The resulting disadvantage is that choosing different parts as symbolic may lead to different results. [3]

Path Coverage

Classic static symbolic execution explores all possible program paths [3]. However, this leads to a path explosion even for smaller programs with only a few thousand lines of code [10]. So that not all paths have to be traversed, heuristics can be used. For example, in dynamic symbolic execution snapshots of the program state can be taken at certain locations. After a path has been explored, the next path can be started from the location where the snapshot was taken. [3]

Conclusion

In summary, symbolic execution can be performed in many different ways. Each has its advantages and challenges. Ultimately, an analyst chooses the appropriate variant based on their constraints and goals. Morion allows analyzing parts of a program symbolically and follows a hybrid approach. First, it concretely executes a program path to record a trace containing actual program states. This trace is then used as input for various static analyses, such as static symbolic execution. Morion is described in more detail in Section 4.1.

2.3.4 Symbolic Execution and Taint Analysis

Taint analysis, like symbolic execution, is a software analysis technique. It allows tracking the flow of data through a program. This is done by marking ("tainting") selected memory data as "taint sources". All memory which has its state derived from data in previously tainted areas is tainted as well. After the analysis is completed, an analyst can check whether any so-called "taint sink" is tainted. A sink is typically a sensitive program location that could be misused to exploit a vulnerability if influenced by a source, such as a call to another program or a write operation to a security-critical file. [3]

Therefore, unlike symbolic execution, taint analysis only reveals whether a program location can be influenced by another, but not how [3]. However, because taint analysis does not require computing the direct relationship between the input and output values of a given instruction, it scales much better than symbolic execution. Because they are still similar, code used to perform symbolic execution can often be used to perform taint analysis as well [13], while the reverse is not true.

Chapter 3

State of the Art

The previous chapter explained symbolic execution and its different variants and challenges. This chapter focuses on the tools that enable symbolic execution: symbolic execution engines. It then introduces some existing user interfaces that simplify the use of symbolic execution engines.

3.1 Introduction to Symbolic Execution Engines

A symbolic execution engine is a piece of software that can perform symbolic execution. That is, an engine builds the symbolic state for program paths by computing the symbolic expression store and path constraints, and by recording the mappings of variables to symbolic expressions. [3] Developing such software is not trivial because every executed instruction and its runtime has to be modelled. For this reason, there are many open-source symbex engines that are designed to serve as a framework or library for building custom tools that leverage symbex.

This work focuses only on symbex engines that provide binary-level symbex capabilities since Morion is designed to analyze binaries. The most popular engines that meet this requirement are angr [14], Triton [15], Manticore [16], S2E [17], Mayhem [18], and BINSEC [19]. Introducing each of these engines is beyond the scope of this paper. However, angr and Triton will be presented in more detail, since their corresponding Graphical User Interfaces (GUIs) can be compared to Ghidion. In addition, Morion is based on Triton.

3.1.1 angr

angr [14] is a binary analysis framework that was created because binary analysis tools often do not go beyond the state of a research prototype. As a result, many contributions to the field were wasted, and researchers often had to repeat implementation work in particular. Therefore, angr implements many modules needed for state-of-the-art binary analysis. [14]

Architecture and Features

Shoshitaishvili et al. [14] developed angr with the goal of “systematizing the field and encouraging the development of next-generation binary analysis techniques by implementing, in an accessible, open, and usable way, effective techniques from current research efforts. . .”. To achieve this, four specific design goals were defined,

namely cross-architecture support, cross-platform support, support for different analysis paradigms, and usability. [14]

angr fulfils these goals by providing strictly separated submodules that allow comparison and composition between different binary analysis techniques. For example, angr achieves the first goal in its "Intermediate Representation" submodule by supporting 32-bit and 64-bit binaries of x86, ARM, MIPS, and PowerPC computer architectures. The second goal was met thanks to the Binary Loading submodule, which is capable of loading Windows, Linux, and FreeBSD binaries. By implementing additional submodules, angr also supports different analysis paradigms. Finally, usability is ensured by the open source implementation, written almost entirely in Python, and its concise API, usable from an interactive IPython shell or as a Python module. [14]

Dynamic symbolic execution is one of the binary analysis techniques that build on angr's submodules and has been fully integrated from the beginning. To solve the resulting path constraints, angr uses the Z3 [12] SMT solver by default, but the implementation of other SMT solvers is easily possible. [14]

Current Limitations

Although angr advertises cross-architecture support, there are some exceptions to be aware of when analyzing binaries from different architectures. In addition, angr has difficulty detecting and modelling library calls from statically linked binaries. Cheng [20] further criticizes that "there is a steep learning curve before one can use it effectively".

3.1.2 Triton

Originally, Triton [15] was primarily a dynamic symbolic (or concolic) execution engine. Over time, however, it evolved into a more general open-source dynamic binary analysis library that simplifies creating program analysis tools and automating reverse engineering, among other things [21]. The following section shows how Triton accomplishes this by taking a look at its architecture and features.

Architecture and Features

Triton provides components such as a symbolic execution engine, a taint analysis engine, or a constraint (SMT) solver interface. It supports the x86, x86-64, ARM32, and AArch64 instruction sets by representing their semantics through abstract syntax trees (ASTs). [22] According to Andriess [3], Triton is best known for its symbolic execution engine and provides APIs for C/C++ and Python.

Triton supports both static (SSE) and dynamic (DSE) symbolic execution through its symbolic emulation and concolic execution modes, respectively. To reduce the risk of path explosion, parts of the symbolic state can be concretized in both modes. The concolic execution mode is faster than the symbolic emulation mode because it only has to compute the symbolic state and gets the concrete state through the concrete execution itself. However, in the concolic execution mode, Triton must always run through the program from the beginning, while the symbolic emulation mode allows analysing of parts of a program. [3] Finally, Z3 [12] or Bitwuzla [23] will solve the path constraints [22].

Current Limitations

Symbolic executions with Triton run primarily offline, i.e. without parallel exploration of paths. However, Triton provides a built-in snapshot mechanism so that overlapping paths do not have to be traversed multiple times. [3]

In addition, Triton directly processes machine instructions. So for each instruction to be executed symbolically, the effect of the instruction on the symbolic state must be provided manually [3]. To address this issue, work is being done on TritonDSE [24], which will add exploration capabilities to Triton.

3.1.3 Conclusion

Symbolic execution engines create the symbolic state of a program path and then solve the path constraint using a constraint solver such as Z3. The way the symbolic state is computed can vary greatly from engine to engine. Often, symbolic execution engines even offer several modes, each with its strengths and weaknesses.

3.2 Symbolic Execution Graphical User Interfaces

Most symbolic execution engines provide access to their functionality through one or more APIs. While this is very powerful, there are cases where it would be more user-friendly if the engine's functionality were additionally accessible via a Graphical User Interface (GUI). This is specifically the case when an analyst examines a program through the user interface of a binary analysis platform such as Ghidra [2], IDA Pro [25], or Binary Ninja [26]. Below, some tools are introduced that have recognized this problem, try to solve it, and have at least some similarities with the result of this paper. In addition, this section presents AngryGhidra [27] in more detail using the `strlen` example (see Listing 1.1), so that a better impression of how these tools work will be gained.

3.2.1 AngryGhidra

AngryGhidra [27] is an open-source extension for Ghidra that allows an analyst to use `angr` for symbolic execution from Ghidra's GUI. AngryGhidra was developed to bring symbolic execution to Ghidra. Therefore, `angr` was chosen as the underlying symbolic execution engine because it is popular and has a well-documented API. [28]

Features

AngryGhidra collects the parameters described below through its window in Ghidra and then uses a fixed and predefined `angr` script [29]. Skliarova [30] describes the parameters as follows:

- `Auto load libs` - If this option is set, `angr`'s loader "determines which shared objects are needed when loading binaries" [31].
- `Blank State` - The optional address from which to start symbolic execution. If no `Blank State` address is set, the entry state of the program is used [27].
- `Find address` - The desired target address when executing the binary.

- **Avoid addresses** – Addresses that are located in branches that are of no interest and are to be avoided during symbolic execution. When they are found, angr will not continue exploring this branch. This might speed up the symbolic execution.
- **Arguments** - Allows supplying a list of values as the program's argument vector (argv).
- **Hook options** - Allows an analyst to intercept specified instructions and enter certain values into the registers.
- **Store symbolic vector** - Allows to make values at certain memory addresses symbolic. If the symbolic execution is successful, the plugin outputs the generated solution vector.
- **Write to memory** - Allows to set specific values at certain memory addresses.
- **Registers** - Allows to initialize registers with specific values.

Limitations

As mentioned above, AngryGhidra uses a fixed and predefined angr script that takes parameters from the UI [29]. Consequently, the extension provides only a subset of angr's symbolic execution capabilities [32].

In addition, usage documentation is essentially non-existent [32]. A Russian article [30] written by the developer of AngryGhidra, which does not seem to be referenced anywhere, appears to be the only source where all the extension's functionalities are explained. Otherwise, the source code of AngryGhidra and the documentation of angr have to be used to discover the meaning of the extension's parameters.

Finally, AngryGhidra only supports the x86 family of architectures. This is unfortunate, as both angr and Ghidra support a variety of binary formats. [32]

Solving a Simple Example

To demonstrate AngryGhidra, Listing 1.1 was compiled using the GNU Compiler Collection (GCC). The resulting binary is imported into a Ghidra project, and then a symbolic execution is performed using AngryGhidra. The corresponding installation instructions can be found in Appendix 1. As a reminder, the main method of `strlen.c` is shown in Listing 3.1. Note that `BUF_LENGTH` is set to 8. The goal of the symbolic execution is to make the length of `s` equal to two so that lines 19 and 20 are reached.

After importing the binary to Ghidra and opening its CodeBrowser, the first task is to identify the main function and set the first instruction after the call to `calloc` as the Blank State address, as seen in Figure 3.1. In this case, the Blank State is set to address `0x00400588`.

As discussed, the goal is that `s` has a length of two. Therefore, the call of the first `printf` will be set as the Find address. Additionally, the call of the second `printf` will be set as an Avoid address. AngryGhidra colours the Find Address green and all the Avoid Addresses red as shown in Figure 3.2.

```
11 int main(int argc, char *argv[]) {
12     char *s;
13
14     // Buffer, will be (partially) symbolised
15     s = (char *) calloc(BUF_LENGTH, sizeof(char));
16
17     // Testing strlen
18     if(strlen(s) == 2) {
19         printf("strlen('%s') == 2\n", s);
20         return EXIT_SUCCESS;
21     }
22     printf("strlen('%s') != 2\n", s);
23     return EXIT_SUCCESS;
24 }
```

LISTING 3.1: Main Method of `strlen.c`

When opening AngryGhidra's panel through Window → AngryGhidraPlugin, the symbolic execution can be started by clicking Run. Right now, no solution will be found. First, the allocated buffer has to be made symbolic, because the symbolic state will be empty otherwise. This example assumes that the buffer of length 8 is allocated at address 0x00412190 to 0x00412197. Therefore, the value 0x00412190 must be additionally stored in register r0, because in ARM architectures return values are stored in register r0. Otherwise, angr would not know where this buffer would be allocated.

The next step is to symbolize the buffer. The analyst symbolizes the entire buffer by storing a symbolic vector of length eight at address 0x00412190 as shown in Figure 3.3. A click on Run will execute the instructions symbolically. After a few seconds, AngryGhidra will report a solution:

```
0x412190 = b'\x01\x01\x00\x00\x00\x00\x00\x00'
```

AngryGhidra reports that in a concrete execution, addresses 0x00412190 and 0x00412191 can be set to 0x01 and all the other addresses can be set to 0x00 to reach lines 19 and 20 in Listing 3.1. This is only one of many possible models (solutions). Because strings are null-terminated in C programming, the effective length of the string will be two. As a result, lines 19 and 20 will be reached in a concrete execution.

3.2.2 angr Management

angr Management [33] is the official GUI for angr and is currently being rapidly expanded by the community. It allows an analyst to easily load a binary and provides several views to inspect it.

Features

The main features in angr Management are the following views:

- **Functions:** Lists all the functions found in the binary and some related information such as the address of the entry instruction.
- **Disassembly:** Translates the binary into assembly code and displays a function graph.

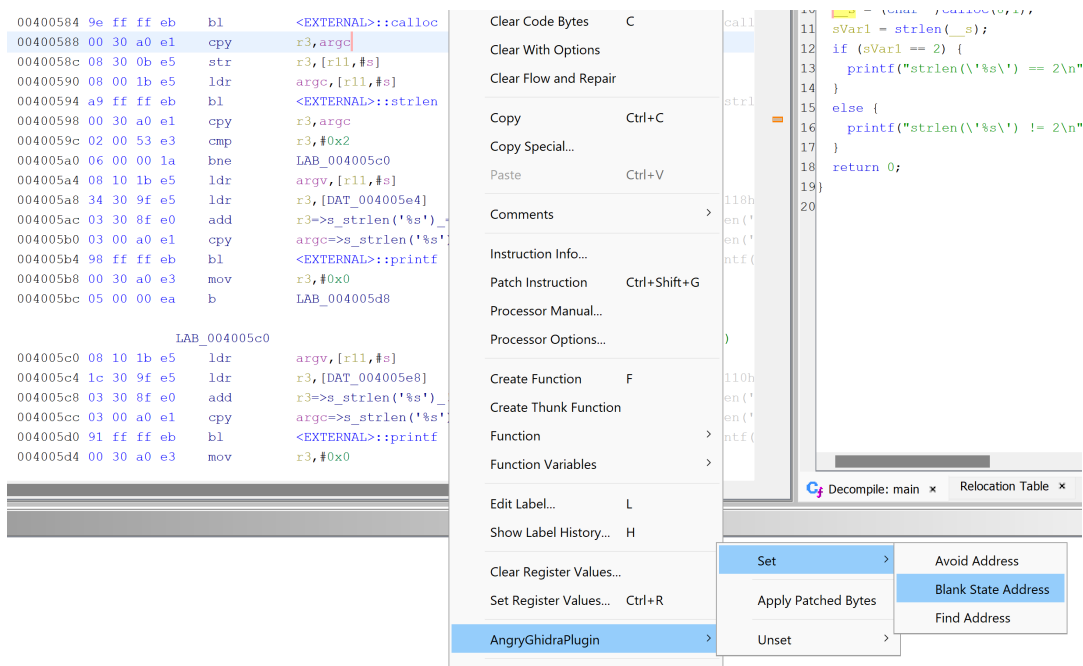


FIGURE 3.1: Set the First Instruction After the Call to `calloc` as Blank State Address Using AngryGhidra

- Hex: Displays the contents of the binary in hexadecimal and ASCII.
- Pseudocode: Shows decompiled pseudocode, similar to the C programming language.
- Symbolic Execution: Allows to symbolically execute the binary (or parts of it).

In addition, angr Management provides an integrated IPython interactive shell that allows an analyst to use the full power of angr interactively. It is worth mentioning that angr Management officially supports the development of plugins.

Limitations

The official angr documentation itself states that the documentation for angr Management, as well as the API for developing plugins, are highly in-flux [34]. However, the documentation for angr itself is more detailed and at least helps to understand the GUI of angr Management.

3.2.3 Ponce

Ponce [35] is a plugin for the popular binary analysis tool IDA Pro [25]. Ponce is designed to eliminate the need for end users to implement specific use cases and advertises that symbolic execution is just one click away. It runs natively on Windows, Linux and OSX and supports the x86, x86-64, Arm, and Arm64 computer architectures. [36]


```

004005a0 06 00 00 1a    bne     LAB_004005c0
004005a4 08 10 1b e5    ldr     argv,[r11,#s]
004005a8 34 30 9f e5    ldr     r3,[DAT_004005e4]
004005ac 03 30 8f e0    add     r3=>s_strlen('%s')==_2_004006cc,pc,r3
004005b0 03 00 a0 e1    cpy     argc=>s_strlen('%s')==_2_004006cc,r3
004005b4 98 ff ff eb    bl     <EXTERNAL>:printf
004005b8 00 30 a0 e3    mov     r3,#0x0
004005bc 05 00 00 ea    b      LAB_004005d8

                                LAB_004005c0                                XREF[1]:
004005c0 08 10 1b e5    ldr     argv,[r11,#s]
004005c4 1c 30 9f e5    ldr     r3,[DAT_004005e8]
004005c8 03 30 8f e0    add     r3=>s_strlen('%s')!=_2_004006e0,pc,r3
004005cc 03 00 a0 e1    cpy     argc=>s_strlen('%s')!=_2_004006e0,r3
004005d0 91 ff ff eb    bl     <EXTERNAL>:printf
004005d4 00 30 a0 e3    mov     r3,#0x0

```

FIGURE 3.2: The Marked Find and Avoid Addresses Using AngryGhidra

Capabilities

Ponce provides taint analysis and symbolic execution capabilities at the binary level. It relies on Triton to perform symbolic execution. Ponce's documentation lists four different use cases. First, it can aid in exploit development by allowing developers to easily see which memory areas and registers can be controlled. Second, during a malware analysis, Ponce can symbolize known commands (specific instructions that malware can execute) and thus understand the conditions under which these commands are executed. Third, it can be helpful in protocol reversal by identifying magic numbers or headers needed. Finally, Ponce simplifies reverse engineering binaries when solving CTFs. [36]

Limitations

One limitation is caused by the use of Triton, which is based on concolic execution. For example, if a symbolic value is used as an index to access non-symbolic data, Triton loses symbolic tracking. Consequently, the originally symbolic value will no longer be symbolic after the data is accessed. [36] Using Triton further means that Ponce is unable to perform static analysis [29].

Another limitation of Ponce is due to IDA, which only offers a very limited variant as freeware, appropriately named IDA Free. Even the version for reverse engineering hobbyists currently costs several hundred US dollars per year. [25] As a result, Ponce is not easily accessible and was not tested by the authors of this paper.

3.2.4 Manticore User Interface

Manticore User Interface (MUI) [37] is a graphical user interface plugin for Ghidra and Binary Ninja that uses the Manticore symbolic execution engine [16]. Manticore is open source and supports dynamic symbolic execution for programs in both traditional and exotic execution environments [38]. In the following sections, the GUI component is referred to as MUI and the symbolic execution engine is referred to as Manticore.

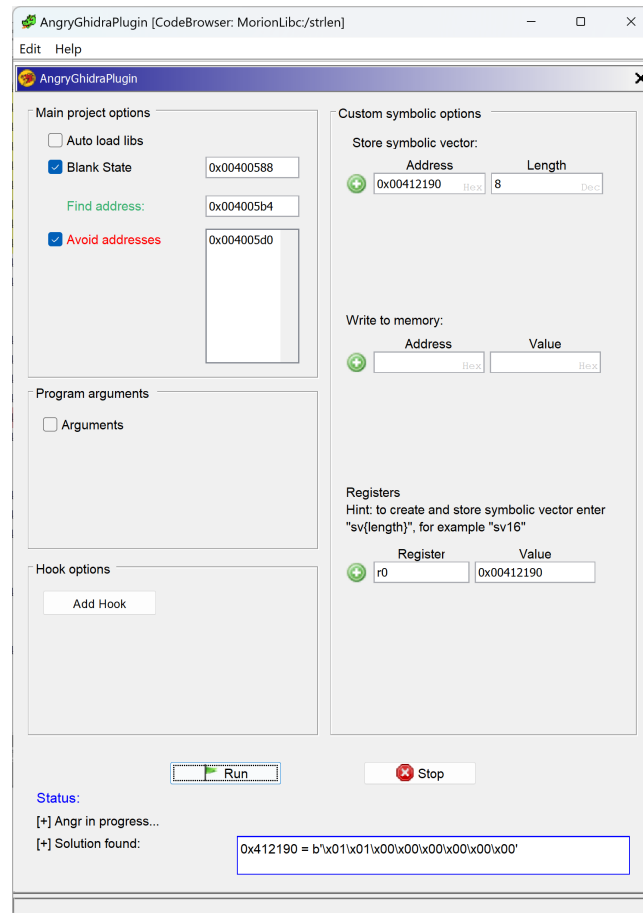


FIGURE 3.3: The Filled AngryGhidra Panel Reports a Solution After Symbolic Execution

Capabilities

The heart of Manticore is its Core Engine, which “implements a generic platform-agnostic symbolic execution engine that makes few assumptions about the underlying execution model” [38]. This gives Manticore the ability to parse not only binaries from traditional computing architectures such as x86, x86-64, or ARMv7 but also Ethereum [39] smart contracts and WebAssembly [40] modules [41]. Initially, Manticore provided only a command-line interface (CLI) and a Python API to access its features and perform custom analysis [41]. To improve its usability, MUI was developed, a user interface for Binary Ninja and Ghidra [37].

MUI has features like setting find and avoid addresses, similar to AngryGhidra (see Section 3.2.1). Further exists both a State List and a Graph View widget. The State List shows all the symbolic states Manticore has explored and how it arrived at a possible solution. The Graph View displays a tree structure of all symbolic states. In addition, it is possible to write custom hooks in Python to take full advantage of Manticore. MUI even supports basic analysis of Ethereum Smart Contracts. [42]

Limitations

MUI and Manticore have some limitations. First, Manticore only supports Linux ELF binaries. Windows and MacOS binaries are not supported. Second, the development pace of Manticore and MUI seems to have stalled. [41] Finally, the documentation for Manticore is poor. Many chapters and sections, for example on Ethereum smart contracts, are empty. [43]

Chapter 4

Concept and Approach

This chapter gives an overview of Morion, how it works, what it offers and how Ghidrion, the Ghidra plugin proposed in this thesis, interacts with it. Furthermore, this chapter presents the broad conceptual structure of Ghidrion with a description of what each part does.

4.1 Morion

Pfammatter [44] describes Morion as a tool developed by the Cyber-Defence (CYD) Campus [1] based on Triton (see Section 3.1.2) to help an analyst understand the capabilities of a bug. It is currently limited to ARMv7 binaries. Specifically, it is designed to be applied to real binaries using a two-step approach. First, a user prepares and records a trace of a concrete execution of the binary. In a second step, this trace can then be analysed using Morion’s vulnerability analysis tools.

4.1.1 Workflow

When analysing a binary with Morion, an analyst would typically go through three steps, possibly several times. Figure 4.1 provides an overview of these steps, which are explained in the following paragraphs.

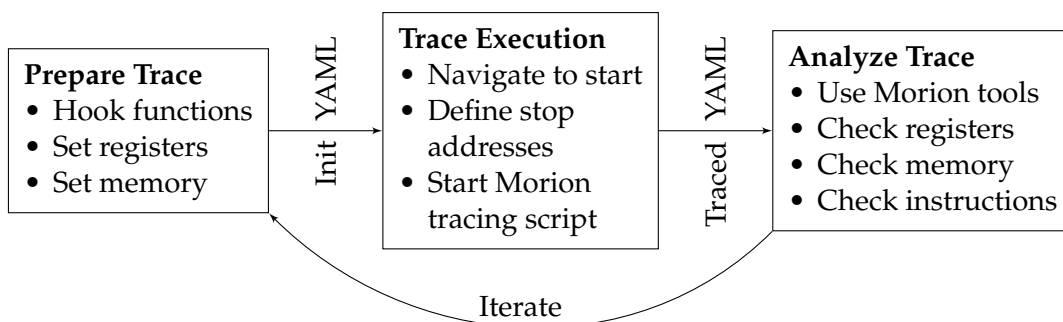


FIGURE 4.1: Workflow When Analyzing a Binary Using Morion.

Preparing the Trace

First, an analyst would prepare the trace. To do this, they may wish to exclude from the trace certain sections of code that either do not affect the potential vulnerability being examined, are too complex and would make the analysis unfeasible due to their complexity, or contain system calls that cannot be modelled using Triton. This can be achieved by using hooks.

A hook consists of an entry address, a leave address, a mode and a replacement implementation. For example, an analyst can hook calls to external libraries (such as `libc`). To do this, they would add a hook where the entry address is that of the jump instruction and the leave address is that of the next instruction after the jump. But the hook system is not limited to this: If the analyst wishes to skip a number of instructions in the analysed code because they are not concerned about the side effects introduced, they can add a hook where the entry and leave addresses are further apart.

Hooks can be of three types: `model`, `taint` and `skip`:

- `model` sections require a substitute implementation within Morion. This will be executed in place of the original instructions whenever the hook is reached. By marking a hook as `model`, parts of the program that add a lot of unnecessary complexity can be greatly simplified. Examples include calls to read a file, memory manipulation functions such as `memcpy`, transformation functions such as `strtol`, or debugging code such as `puts` calls.
- Using hooks of type `taint` allows a taint analysis to be performed on the section of code being analysed using Triton's `sybex` engine. This reduces the complexity of the resulting path constraint SMT formula by marking the results of the hooked function as symbolic but loses the connection between function arguments and return value(s).
- Marking hooks as `skip` causes the engine to skip the hooked instructions. This can be used to ignore sections of code that do not affect the result.

Besides hooks, the analyst may add certain values in registers or at certain memory addresses, which Morion sets just before the tracing starts.

Trace

After preparing the trace, the analyst would start the GNU Debugger (GDB) and manipulate the program to the state they would like it to be in at the start of the trace. To do this, the full functionality of GDB can be used, including breakpoints and both memory and register manipulation.

Morion provides a GDB script that performs the actual tracing. The analyst starts this script and provides it with a list of addresses (breakpoints) that, when reached, stop the trace. These may include the intended target address along with addresses in the binary's error handling code that would stop the trace in case of an error during the trace setup. The script then steps through the program, recording the executed instructions along with additional information such as the instruction address and parameter and result values. When a hook entry address is reached, the tracing script executes the substitute implementation (if the hook is of mode `model` or `taint`) and

continues at the leave address of the hook. It also keeps track of all memory and registers accessed during the trace. For each, the script stores both the value at the start of the trace and the value at the end of the trace.

Analysing the Trace

Using the information gathered during tracing, the analyst can now begin to examine the section of code in question for vulnerabilities. Before performing any analysis, they mark as symbolic registers and memory entries in the entry state that can be influenced by the attacker. Morion provides modules to perform different types of analysis. These modules initialise Triton with the entry memory and register values recorded during tracing. They then execute the recorded instructions symbolically, including model and taint hooks. After executing all instructions, they mark as symbolic all leave memory entries and registers that are influenced by a symbolic entry memory entry or register. The modules also provide an interactive Python shell at each finding, allowing the analyst to perform further manual analysis to verify that the vulnerability is exploitable.

Morion currently provides five analysis modules:

- **Backward Slicing:** This module symbolically executes a program trace for backward slicing. The analysis identifies backward slices for a specified register or memory address. Program slicing allows an analyst to reason about what parts of the program influenced the values of a set of variables at a specified point [45].
- **Control Hijacks:** This module symbolically executes a program trace to identify potential control flow hijacks. A control flow hijack corresponds to registers that affect the control flow, such as the program counter (PC) becoming (partially) symbolic.
- **Memory Hijacks:** This module symbolically executes a program trace to identify potential memory hijacks. A memory hijack corresponds to the target of a memory read or write operation becoming (partially) symbolic.
- **Branches:** This module symbolically executes a program trace for branch analysis. The analysis identifies multi-way branches along the trace and outputs concrete values of how to reach the branch not taken. A given branch is only evaluated once.
- **Paths:** This module symbolically executes a program trace for path analysis. The analysis identifies unique paths along the trace and outputs concrete values of how to reach these paths. A path consists of a sequence of multi-way branches. The last multi-way branch in each output path is not taken in the concrete execution of the trace.

Finally, the analyst may wish to manually analyse the executed instructions, compare the entry and leave states and check for any changes that might indicate a vulnerability or check for any unexpected leave memory entries or registers marked as symbolic.

4.1.2 Interface

Morion stores its intermediate state in YAML files. This allows Ghidrion to interface with Morion without the need to interact with Python code directly from its Java environment. These files typically accumulate information with each step during a typical analysis according to the typical workflow described in Section 4.1.1. And because, strictly speaking, excess information is not illegal, the output of each step is a legal input to run the same step again. In general, one can distinguish between the information stored in the files before and after tracing. In the rest of this thesis, these will be referred to as Init YAML and Traced YAML files (see also Figure 4.1), with the latter containing a superset of the former's information.

Init YAML Files

A typical Init YAML file for the example introduced in Section 1.3 (specifically, in Listing 1.1) might look like what can be found in Listing 4.1. It describes three hooks: The `printf` calls are skipped, and the `strlen` call is replaced by the substitution implementation in Morion. The file does not describe any entry register values but sets the memory values for addresses `0x00412190` through `0x00412197` to the ASCII representation of the letters A to G including the trailing NULL. Three of these entries are already marked as symbolic, as indicated by the `$$`.

```
1 hooks:
2   libc:
3     printf:
4       - entry: '0x4005b4'
5         leave: '0x4005b8'
6         mode: skip
7       - entry: '0x4005d0'
8         leave: '0x4005d4'
9         mode: skip
10    strlen:
11      - entry: '0x400594'
12        leave: '0x400598'
13        mode: model
14  states:
15    entry:
16      mems:
17        '0x00412190': ['0x41']
18        '0x00412191': ['0x42', '$$']
19        '0x00412192': ['0x43', '$$']
20        '0x00412193': ['0x44']
21        '0x00412194': ['0x45', '$$']
22        '0x00412195': ['0x46']
23        '0x00412196': ['0x47']
24        '0x00412197': ['0x00']
```

LISTING 4.1: Example Init YAML File

Traced YAML Files

After a traced execution, the Traced YAML file now has additional entries. The complete Example Traced YAML file can be found in Appendix 2. First, it contains a list of all executed instructions (see Listing 4.2). These entries consist of the instruction address, the instruction in binary and human-readable assembly, and a comment.

Line 33 shows that the hook set in line 10 of the Init YAML file (see Listing 4.1) was correctly recognised and executed.

```

17 instructions:
18   - - '0x00400588'
19     - 00 30 a0 e1
20     - 'mov r3, r0'
21     - 'L15: `s = (char *) calloc(BUF_LENGTH, sizeof(char));`'
22     - - '0x0040058c'
23     - 08 30 0b e5
24     - 'str r3, [fp, #-8]'
25     - 'L15: `s = (char *) calloc(BUF_LENGTH, sizeof(char));`'
26     - - '0x00400590'
27     - 08 00 1b e5
28     - 'ldr r0, [fp, #-8]'
29     - 'L18: `if(strlen(s) == 2) {'`
30     - - '0x00400594'
31     - 59 ff ef ea
32     - 'b #-0x400294'
33     - '// Hook: libc:strlen (on=entry, mode=model)'
34     - - '0x00000300'
35     - 07 00 a0 e3
36     - 'mov r0, #0x7'
37     - '// Hook: libc:strlen (on=leave, mode=model)'

```

LISTING 4.2: Excerpt From the Instructions of the Example Traced YAML File

It further contains an extended entry state with all accessed values in addition to those provided in the Init YAML file. For each entry in the entry state, the value at the end of the tracing is also recorded. Excerpts of this can be found in Listing 4.3.

4.2 Parts of the Plugin

Based on the available interface, Ghidrion can be divided into two parts: The first supports an analyst in creating an Init YAML, while the second helps in analysing the Traced YAML.

4.2.1 Creating an Init YAML File

Here, the plugin supports the analyst by using Ghidra's analysis to find all hookable function calls and present the user with a list from which to choose. In addition, the analyst can add memory entries manually, using convenience features such as splitting multi-byte data and repeating a byte of data across multiple addresses. Registers can be added either manually or by selecting from all available registers. Both registers and memory entries can be marked as symbolic.

This part of the plugin is also able to import existing Init YAML files to continue working on them and export them to files on disk that can then be fed to Morion.

4.2.2 Analysing a Traced YAML File

The second part of the plugin supports the analysis of the information collected during a trace in three ways:

```

107 states:
108   entry:
109     addr: '0x00400588'
110     mems:
111       '0x004005e8': ['0x10']
112       '0x004005e9': ['0x01']
113       '0x004005ea': ['0x00']
114       '0x004005eb': ['0x00']
115       '0x00412190': ['0x41']
116       '0x00412191': ['0x42', '$$]
117       '0xbefffb77': ['0xb6']
118
119     regs:
120       r0: ['0x00412190']
121       r1: ['0x00412190']
122       r11: ['0xbefffb74']
123       r3: ['0x00412198']
124       sp: ['0xbefffb60']
125       z: ['0x00000000']
126
127   leave:
128     addr: '0xb6eed5a0'
129     mems:
130       '0x004005e8': ['0x10']
131       '0x004005e9': ['0x01']
132       '0x004005ea': ['0x00']
133       '0x004005eb': ['0x00']
134       '0x00412190': ['0x41']
135       '0x00412191': ['0x42']
136       '0xbefffb77': ['0xb6']
137
138     regs:
139       r0: ['0x00000000']
140       r1: ['0x00000000']
141       r11: ['0x00000000']
142       r3: ['0x00000000']
143       sp: ['0xbefffb78']
144       z: ['0x00000001']

```

LISTING 4.3: Excerpt From the Entry and Leave State of the Example Traced YAML File. No Leave Entries Are Marked as Symbolic Since No Analysis Was Performed Yet.

1. It marks the executed instructions in Ghidra's standard Listing and Decompile windows, thus allowing an analyst to quickly check which instructions were executed during the trace.
2. It provides a way for an analyst to quickly and easily gain insight into what data and registers have changed during execution. It marks which of these are only present in either the entry or leave state, which have previously been marked as symbolic by the analyst in the entry state, and which of the leave state entries are influenced by the symbolic entry state entries.
3. It launches Morion's analysis modules using the selected Traced YAML. The interactive shell provided during the execution has access to the information about the binary from Ghidra's analysis.

Chapter 5

Architecture and Implementation

The first part of this chapter provides documentation of how Ghidra can be extended, especially about plugins that interact with a loaded binary. The second part explains the technology used and the architecture implemented in Ghidra.

5.1 Extending Ghidra

Although Ghidra offers countless features, there are many problems for which Ghidra does not provide a built-in solution. Therefore, Ghidra supports third-party development of scripts and feature-rich extensions.

5.1.1 GhidraDev

The most obvious way to analyze binaries in ways not supported by Ghidra is to write and edit Ghidra Scripts using Ghidra's Script Manager window. It provides a basic internal scripting environment where existing scripts can be edited and new scripts can be created. However, since this scripting environment does not support auto-completion or debugging, the development experience is limited. Fortunately, Ghidra offers a plugin for the popular Eclipse IDE called GhidraDev that greatly simplifies development.

GhidraDev not only allows a developer to write Ghidra Scripts but also Ghidra Modules. Both allow to extend Ghidra with custom functionality and are briefly described in the next two sections.

5.1.2 Ghidra Scripts

Ghidra scripts can be written in both Java and Python 2. Python 3 is not supported because Ghidra is developed in Java and Ghidra's Java objects are accessed via Jython [46] which only supports Python 2.7. A script written in Java is a complete class extending the abstract `GhidraScript` [47] class which requires the implementation of the `run()` method. Finally, when a script is executed, the `run()` method is called. [48]

Ghidra constructs can be accessed from Ghidra scripts via the so-called Flat Application Programming Interface (Flat API) [49]. The Flat API is a single class called `FlatProgramAPI` that exposes many levels of the hierarchical `Program` [50] API. [48] Ghidra uses the Flat API only in its `TraceColorizerScript` class by extending `GhidraScript`, which in turn extends `FlatProgramAPI`. Otherwise, the `Program` API

is used to interact with Ghidra objects. The key objects of the Program API used by Ghidrion are described in Section 5.1.4.

5.1.3 Ghidra Module Projects

GhidraDev not only allows the creation of Ghidra Scripts but also Ghidra Module Projects. A Ghidra Module Project is a Java project with associated help files and documentation. Moreover, it allows controlling how to interact with other Ghidra modules. [48] When creating a new module, Ghidra asks the developer which module templates to use. Currently, there are six module templates available:

- **Analyzer:** Provides a skeleton class extending `AbstractAnalyzer` [51]. Analyzers allow extending Ghidra's analysis on binaries.
- **Plugin:** Provides a skeleton class extending `ProgramPlugin` [52]. Plugins allow to access Ghidra's GUI and the event notification systems [53]. A plugin is a bundle of features and capabilities that can be enabled and disabled in Ghidra. Plugins expose their features and capabilities via a user interface (UI), so-called service APIs (see 5.1.3), or `PluginEvents` [54] and follow a life cycle. Implementing a `ComponentProvider` [55] allows providing a UI. [56] This template was chosen for Ghidrion development because it allows access to the Program API and the development of a UI.
- **Loader:** Provides a skeleton class extending `AbstractProgramWrapperLoader` [57]. Loaders allow supporting new binary code formats [53].
- **FileSystem:** Provides a skeleton class implementing `GFileSystem` [58]. FileSystems allow supporting more archive files such as apk, ZIP, or tar [53].
- **Exporter:** Provides a skeleton class extending `Exporter` [59]. Exporters allow to export parts of a program [53].
- **Processor:** These modules handle the disassembly operations in Ghidra. It is the only template that does not provide a skeleton class. [48]

Finally, GhidraDev allows exporting a module as a ZIP file with just a few clicks. This file can then be easily distributed and installed in Ghidra.

Ghidra Services

A plugin can expose its capabilities in the form of one or more services. Other plugins can then acquire these services. For example, Ghidrion acquires the `ColorizingService` to colour a Morion trace in Ghidra's Listing window. On the other hand, the `DecompilerHighlightService` is used to highlight the Morion trace in Ghidra's Decompile window.

Discovering what services are available is rather difficult since there does not seem to be a central list of all available services. While some are listed in the `ghidra.app.services` package, the authors found that the easiest way to discover them is by manually searching for other plugins and checking what services they use or provide.

```

1 public static Set<HookableFunction> getHookableFunctions(Program program)
  ↪ {
2     FunctionManager functionManager = program.getFunctionManager();
3     ReferenceManager referenceManager = program.getReferenceManager();
4     Memory memory = program.getMemory();
5     Set<HookableFunction> res = new HashSet<>();
6
7     for (Function externalFunction : functionManager.getExternalFunctions())
8         for (Address thunkAddress :
  ↪     externalFunction.getExternalThunkAddresses(true))
9             for (Reference reference :
  ↪     referenceManager.getReferencesTo(thunkAddress))
10                if (!reference.isEntryPointReference()) {
11                    String name = externalFunction.getName();
12                    Address entryAddress = reference.getFromAddress();
13                    Instruction instruction =
  ↪     program.getListing().getInstructionAfter(entryAddress);
14                    if (instruction == null) // if there is no next instruction,
  ↪     hooking doesn't work
15                        continue;
16                    Address leaveAddress = instruction.getAddress();
17                    res.add(new HookableFunction(name, entryAddress, leaveAddress,
  ↪     memory));
18                }
19     return res;
20 }

```

LISTING 5.1: Method Gathering Calls to External Functions in a Program That Can Be Hooked in Ghidrion

5.1.4 Ghidra's Program API

Ghidrion makes heavy use of Ghidra's Application Programming Interface (API) [60]. Unfortunately, apart from the sparse JavaDoc, there does not seem to be an overview of the API's structure. This chapter presents some classes that the authors of this paper found useful in the development of Ghidrion along with an overview of the functionality they provide to aid in future program plugin development.

Since Ghidrion is a `ProgramPlugin` [52], the most interesting functionality is provided by `Program` [50] objects. These can be obtained by overwriting a function in the main class that is called by Ghidra every time a program is opened or activated [52]. They contain getter functions that return so-called managers which then provide functionality regarding a certain part of the loaded binary. Listing 5.1 shows how some of those managers are used to gather all external function calls in a certain `Program`. The following sections present a selection of these managers and explain how they are used in Ghidrion.

FunctionManager

The `FunctionManager` [61] object lies at the core of the external function hooking capability of Ghidrion (see Section 4.1.1). To display a list of all external functions along with all their references in the code, Ghidrion first obtains a list of all external functions using the `FunctionManager` including their names and addresses in the Procedure Linking Table (`.plt`). Then, it generates a list of the addresses of thunks to those external functions to get the references in the Global Offset Table (`.got`).

Finally, using the `ReferenceManager`, all references to either of the above are found. These typically are in the text section (`.text`) of the executable.

ReferenceManager

The `ReferenceManager` [62] object provides all references that Ghidra detects (such as function calls or constants), thus allowing plugins using it to leverage the referencing engine of Ghidra. It works on an address basis, which requires back-and-forth translating between `Address` and `Function` objects at times, which can be done using the `FunctionManager`.

Memory

The `Memory` [63] object contains information about the memory contents and layout. Ghidra only uses it to map addresses of function references to their ELF sections to allow easier filtering while adding function hooks.

ProgramContext

The `ProgramContext` [64] object allows interfacing with the registers a machine capable to execute the loaded binary must have. It can be used to get a list of all such registers, including their attributes such as their size and type.

5.1.5 Context Menus in Ghidra

To create a context menu in the main Ghidra window such as the program Listing, a developer can register a context action during the plugin setup. These contain 4 methods that can be overridden to define what should happen if they are selected and in which contexts and on which lines they should be displayed and enabled. Ghidra uses this to allow an analyst to hook functions directly from within Ghidra's Listing window with a right click. To achieve this, it registers three types of `ListingContextAction` [65]:

1. It allows an analyst to add a hook using all possible hooking modes. This context action is only available if the selected line contains a reference to an external function and said function is not yet added. The code to create this type of `ListingContextAction` can be seen in Listing 5.2.
2. If a hook has previously been added using either the context menu or Ghidra's main window, it allows the analyst to change the mode of the hook.
3. It allows the analyst to delete the hook associated with a certain line, should it exist.

5.2 Architecture and Technology

This section discusses the technology Ghidra is built on and architectural principles followed during the implementation.

```

1 private ListingContextAction getAddHookAction(Mode mode, Program program)
  ↪ {
2     return new ListingContextAction(LISTENING_CONTEXT_ACTION_NAME,
  ↪     getName()) {
3         @Override
4         protected void actionPerformed(ListingActionContext context) {
5             Address entryAddress = context.getLocation().getAddress();
6             Address leaveAddress = program.getListing()
  ↪             .getInstructionAfter(entryAddress).getAddress();
7             Optional<Function> function = getFunctionAtSelectedLocation(context,
  ↪             program);
8             String name = function.get().getName(); // checks are done in
  ↪             isValidContext
9             String libraryName = JOptionPane.showInputDialog("Input library
  ↪             name", "libc");
10            traceFile.getHooks().add(new Hook(libraryName, name, entryAddress,
  ↪            leaveAddress, mode));
11        }
12
13        @Override
14        protected boolean isValidContext(ListingActionContext context) {
15            Address address = context.getLocation().getAddress();
16            Optional<Function> function = getFunctionAtSelectedLocation(context,
  ↪            plugin.getCurrentProgram());
17            return function.isPresent()
18                && function.get().isExternal()
19                && traceFile
20                    .getHooks()
21                    .stream()
22                    .filter(hook -> hook.getEntryAddress().equals(address))
23                    .count() == 0;
24        }
25    };
26 }

```

LISTING 5.2: Method Creating the Context Action To Add Hooks
From Ghidra's Listing Window

5.2.1 Java/Swing

Ghidra is written in Java and its API is natively accessible using Java [60]. Ghidrion is also written in Java to make interaction with the API as seamless as possible. For the graphical user interface (GUI), Ghidra uses Java Swing and provides a way for a plugin to create a window by extending `ComponentProvider` [55]. These then behave like any other Ghidra window and can be filled with any combination of Swing elements.

5.2.2 Model-View-Controller Pattern

Ghidrion is written using the Model-View-Controller (MVC) pattern. According to Burbeck: "In the MVC paradigm the user input, the modelling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task." [66] This enables separation of logic and UI code that minimises object dependencies. Swing itself is also rooted in the MVC design [67].

5.2.3 Observer Pattern

In Ghidrion, the observer pattern (as described by Gamma et. al [68]) allows different types of interfaces (such as the main GUI, the context menu, and the colouring of visited instructions) to work with the same information, thus allowing synchronisation between their states. For example, the context menu relies on the same object to manipulate hooks as the main GUI, without the need for direct references between interface objects. It further allows objects to subscribe to changes in data and perform actions based on the new data. This is used to update the GUI whenever a user changes something in the internal representation of either YAML file using any of the available interactions, and more generally to store whatever information can be entered by the user that has an immediate effect without further interaction. Examples include the colour chosen to highlight executed instructions and the cascading updates in the hook filters.

5.2.4 Graphical User Interface Design

For the GUI itself, the authors chose to rely heavily on Swing's `JTabbedPane`, as it allows the user to see all the necessary information even on small screens. Certain parts of the plugin required an intimate understanding of Swing's internals, such as the custom table renderer used in the diff view, which is needed to highlight table entries by colour. Ghidrion also contains certain custom components made from standard Swing components, such as the filter elements used to filter the hooks to be added.

5.2.5 Python Integration

In Section 4.2.2 it was stated that the plugin starts an interactive Python shell that can access both Morion's and Ghidra's analysis modules. As mentioned in Section 5.1.2, Morion was written in Python 3 while Ghidra only supports Python 2.7. Therefore, solutions were sought to overcome this problem. Two possible solutions came in the form of third-party plugins, Ghidrathon [69] and Pyhidra [70].

Ghidrathon "adds Python 3 scripting capabilities to Ghidra" [69]. It fully replaces the existing Python 2 support and is intended to make it easier to use existing Python 3 tools in Ghidra. In addition, the developers of Ghidrathon mention that they put a lot of emphasis on enabling third-party modules. [69] Pyhidra on the other hand "provides direct access to the Ghidra API" [70] through CPython 3. Among other things, it features an interpreter window that supports Python 3 [70].

However, the authors found that the Morion analysis modules are not fully functional with either Ghidrathon or Pyhidra. Although the import of Morion works, both plugins throw errors when calling Python's `input()` function. However, the proper working of `input()` is necessary to fully use the Morion analysis modules.

Chapter 6

Results

This chapter presents the contributions of this thesis, starting with the created documentation and how this supports future plugin development. Second, it presents the functionality achieved in Ghidra and how it helps an analyst to prepare an Init YAML file. Third, this chapter gives an overview of how a Traced YAML file can be analysed and finishes with what could not be implemented.

6.1 Created Documentation

The contribution of this thesis when it comes to documentation is twofold: First, Section 5.1 provides a missing overview of and introduction to different ways of extending Ghidra, including the used tooling and starting points for future development. Second, it provides a well-designed and well-documented example of a `ProgramPlugin`, which is going to be helpful to developers creating similar plugins.

6.2 Creating an Init YAML File

As mentioned in the concept presented in Section 4.2.1, the first part, located in the first tab of Ghidra, supports an analyst in creating an Init YAML file in three ways: It allows them to add hooks for external functions, memory entries, and registers. This section describes in detail how these objectives were achieved.

6.2.1 Adding Hooks

The first sub-tab allows an analyst to add function hooks to the Init YAML file (see Figure 6.1). The library of the function must currently be entered manually. Using a set of cascading filters, the analyst can then filter all external function calls by function name, memory block name, and hook address. The items within any of these categories can be filtered using a text filter field capable of parsing regular expressions. All subsequent filter lists only show possible hooks that pass the previous filter. By selecting one or more items from the list of items that pass the text filter, each filtering step can be further refined. The analyst can then select the hook mode they wish to use and add all the hooks that pass the filters. If a new hook has been added previously, it will be overwritten using the new mode.

Hooks can additionally be added and deleted, and an analyst can change their mode using the (right click) context menu in Ghidra's Listing window (see Figures 6.2

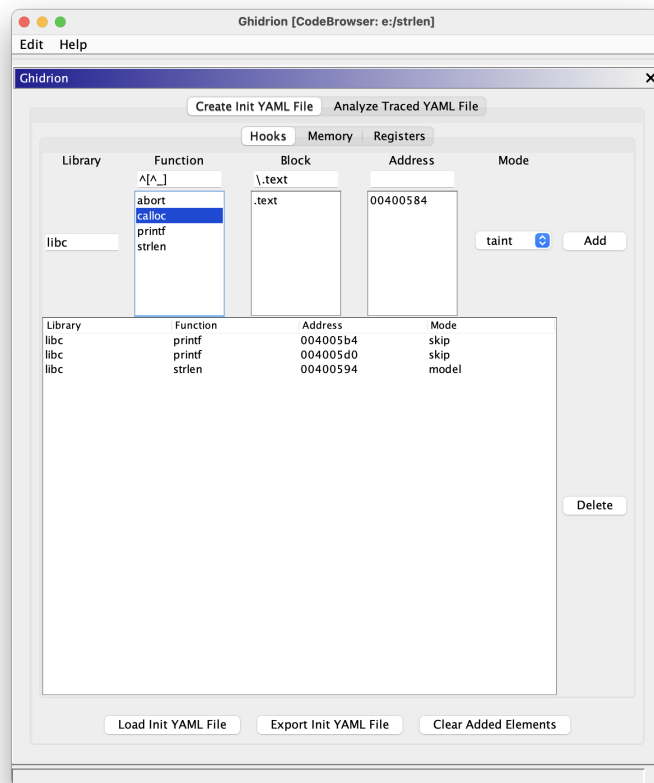


FIGURE 6.1: Ghidridion Tab Allowing Analysts To Filter and Add Hooks for External Functions

and 6.3). Hooks added either way will appear in a list in Ghidridion's main window, from where they can be deleted.

6.2.2 Adding Memory

Figure 6.4 shows the sub-tab where an analyst can add memory entries to the entry state of an Init YAML file. It behaves differently depending on the information provided:

- If no end address is specified and the value is at most one byte (i.e. two hexadecimal characters), an entry is created with the specified value at the specified address.
- If an end address greater than the start address and a value of at most one byte is provided, the value is repeated for every address between and including the start and end addresses. This can be used, for example, to set an entire memory range as symbolic.
- If no end address is specified, but the value is larger than one byte, the value is split into entries with incremental addresses.
- If none of the above applies, an error is displayed.

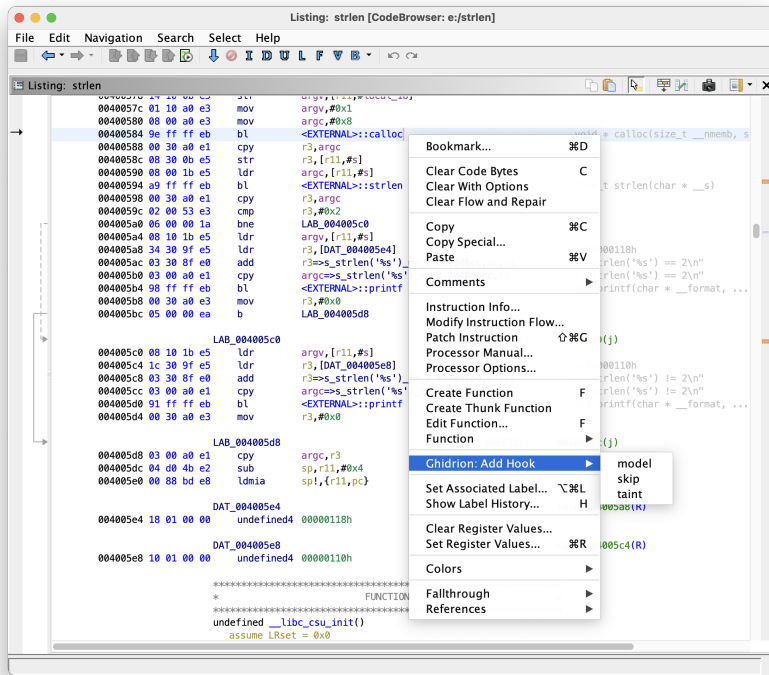


FIGURE 6.2: Adding a Hook Using the Context Menu in Ghidra's Listing Window

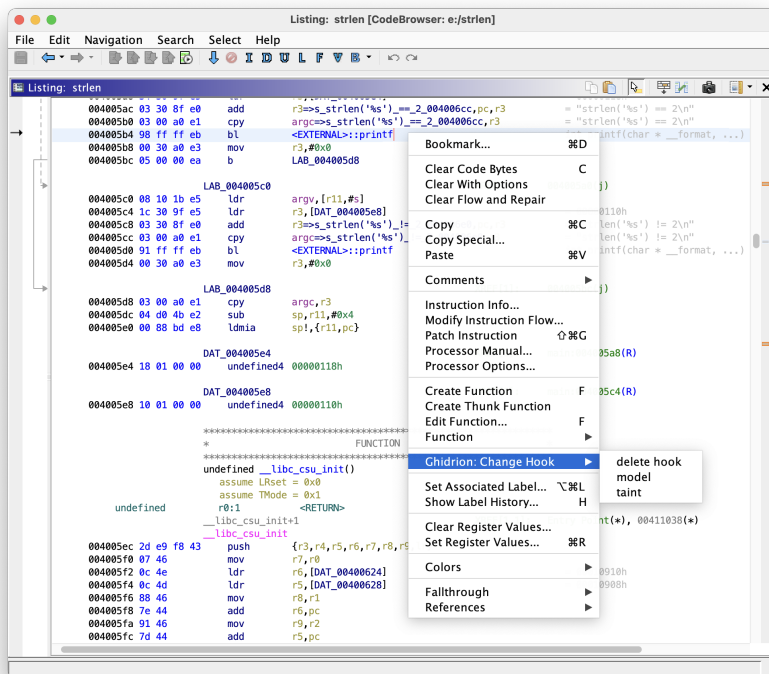


FIGURE 6.3: Changing a Hook Using the Context Menu in Ghidra's Listing Window

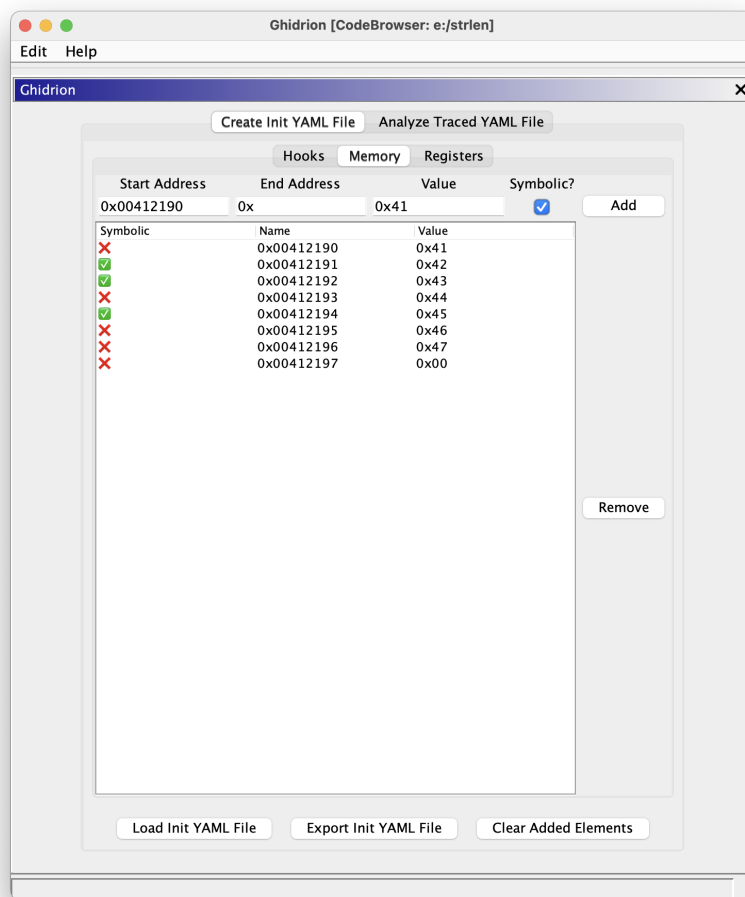


FIGURE 6.4: Ghidion Tab Allowing Analysts To Add Memory Entries

The table below the entry section contains a list of all memory entries previously added to the Init YAML file. One or more entries can be selected and removed. Adding a memory entry with the address of a previously added entry replaces it.

6.2.3 Adding Registers

Adding registers to the Init YAML file is very similar to adding memory entries. The analyst specifies a register name, up to four bytes for the value, and whether the register should be marked as symbolic or not. Existing entries with the same name as the register to be added get replaced. Analogous to the tab for adding memory entries, there is a table for checking and removing previously added registers, as shown in Figure 6.5.

6.2.4 Saving and Importing Init Trace Files

After adding hooks, memory entries and registers, the internal representation of the Init YAML file can be exported to a file on disk, which is then compatible with Morion and allows an analyst to create a trace. If they wish to continue editing a previously created Init YAML file, they can load it into the plugin to do so.

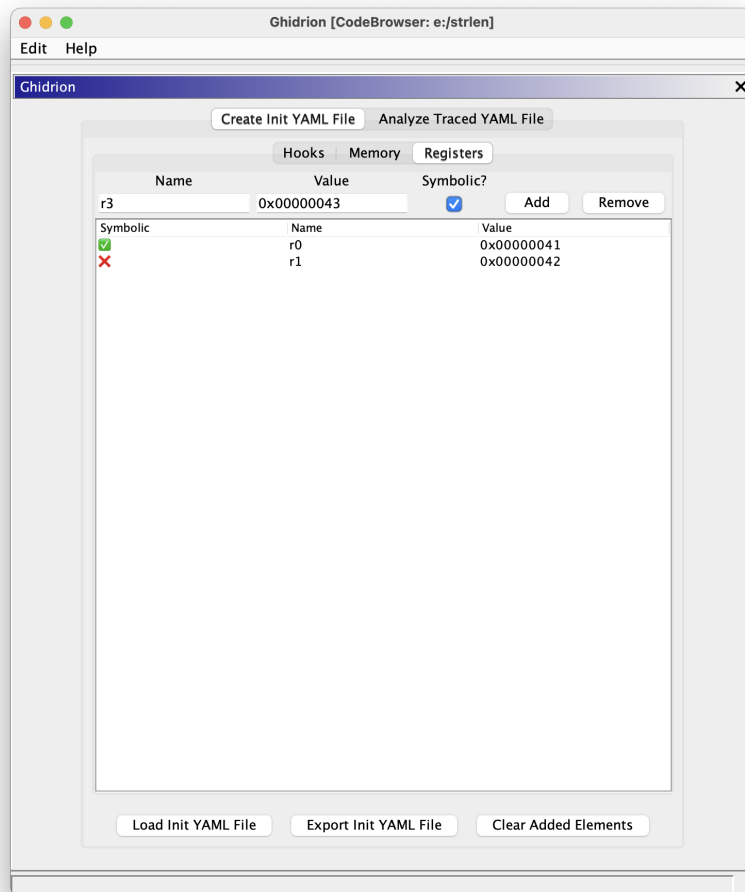


FIGURE 6.5: Ghidrion Tab Allowing Analysts To Add Registers. The Register Entries in the Table in the Lower Half Differ From the Example Init YAML File Introduced in Section 4.1.2.

6.3 Tracing

Tracing has to be done in a GDB environment. This can be either in an external shell or using Ghidra's Debugger tool. Ghidrion does not affect this part of Morion's workflow (see Figure 4.1).

6.4 Analysing a Trace

The concept (see Section 4.2.2) describes that analysing a Traced YAML file consists of comparing the entry and leave states, looking at the recorded instructions and using Morion's analysis tools. The second tab of Ghidrion provides this functionality.

6.4.1 Differences Between Entry and Leave States

Ghidrion's analysis tab has two sub-tabs. These provide a way to quickly check which memory and register values changed during the instructions recorded in the trace, and which of these were marked as symbolic by Morion's analysis tools. The table entries are printed in different colours depending on certain characteristics, as shown in Figures 6.6 and 6.7:

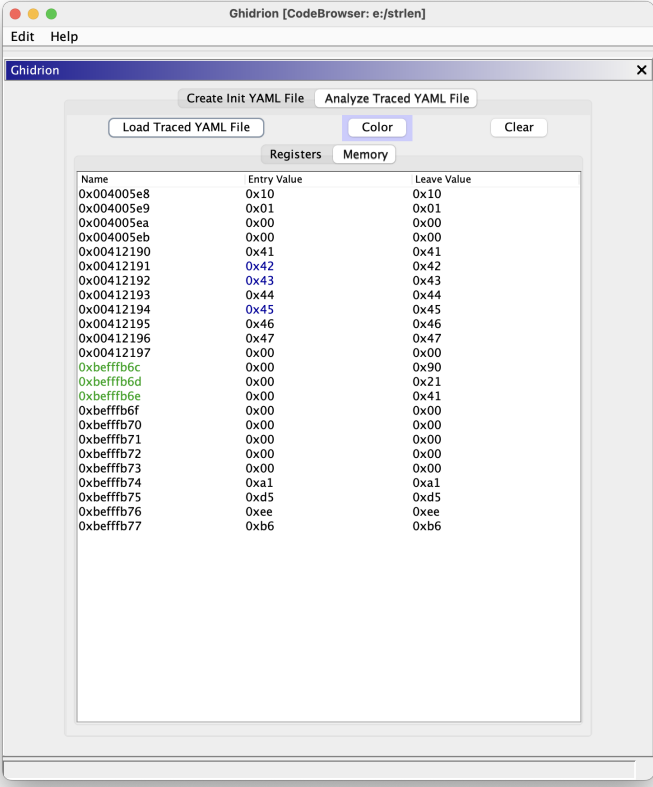
- The register name or memory address is printed in green ink if the values are different, and in red ink if the value is missing in either the entry or leave state.
- If a value is marked as symbolic in either the entry or leave state, the corresponding table entry is printed in blue ink.
- If none of the above applies, the cell values are printed in black ink.

6.4.2 Marking Visited Instructions

When a Traced YAML file is imported, Ghidrion marks all recorded instructions in Ghidra's Listing and Decompile window. This allows an analyst to easily see which parts of a binary were executed during tracing, as shown in Figure 6.8. The colour of the marks can be selected using a colour picker and removed using the corresponding button.

6.4.3 Using Morion's Analysis Modules

Due to the reasons listed in Section 5.2.5, Ghidrion does not currently support using Morion's analysis modules within its GUI. This must be done manually in an external shell.



The screenshot shows the Ghidridion application window with a table of memory values. The table has three columns: Name, Entry Value, and Leave Value. The 'Entry Value' column is highlighted in blue, and the 'Leave Value' column is highlighted in green. The 'Name' column contains hexadecimal addresses. The 'Entry Value' column contains hexadecimal values, and the 'Leave Value' column contains hexadecimal values. The 'Color' button is highlighted in blue.

Name	Entry Value	Leave Value
0x004005e8	0x10	0x10
0x004005e9	0x01	0x01
0x004005ea	0x00	0x00
0x004005eb	0x00	0x00
0x00412190	0x41	0x41
0x00412191	0x42	0x42
0x00412192	0x43	0x43
0x00412193	0x44	0x44
0x00412194	0x45	0x45
0x00412195	0x46	0x46
0x00412196	0x47	0x47
0x00412197	0x00	0x00
0xbeffff6c	0x00	0x90
0xbeffff6d	0x00	0x21
0xbeffff6e	0x00	0x41
0xbeffff6f	0x00	0x00
0xbeffff70	0x00	0x00
0xbeffff71	0x00	0x00
0xbeffff72	0x00	0x00
0xbeffff73	0x00	0x00
0xbeffff74	0xa1	0xa1
0xbeffff75	0xd5	0xd5
0xbeffff76	0xee	0xee
0xbeffff77	0xb6	0xb6

FIGURE 6.6: Differences in Memory Values Before and After Tracing

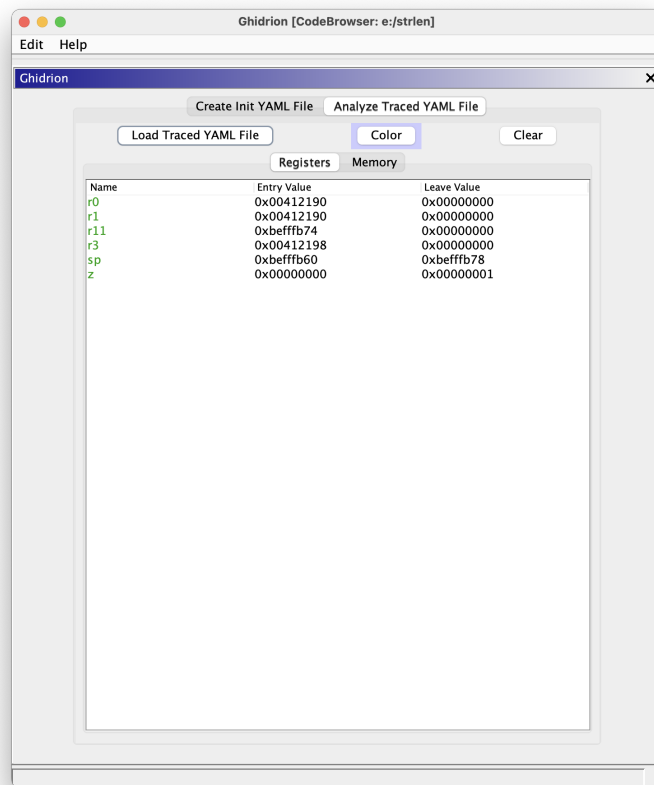


FIGURE 6.7: Differences in Register Values Before and After Tracing

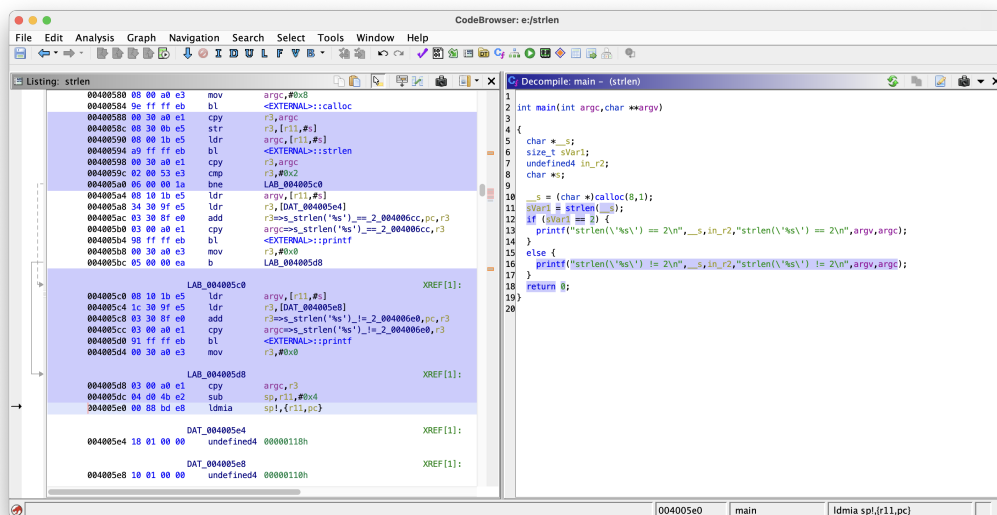


FIGURE 6.8: Executed Instructions During a Trace Marked in Ghidra's Listing and Decompiler Windows

Chapter 7

Discussion and Outlook

This chapter discusses how the results described in Section 6 may impact both future plugin developers and analysts using Ghidrion. Additionally, it looks at known issues and ideas that may be addressed in future work.

7.1 Created Documentation

Unfortunately, the documentation available for developing Ghidra plugins is very sparse. Ghidra itself only publishes instructions on how to install GhidraDev [71]. Its API documentation consists only of JavaDoc, which provides little detail and no overview. Additional information is available in books by Eagle and Nance [48] and David [53], but they are mostly focused on using Ghidra or developing Ghidra modules other than ProgramPlugins (such as Ghidrion). The only other source of information available to the authors were other open-source plugins (especially regarding service discovery, as described in Section 5.1.3). However, they often contain little or no documentation.

This slowed down the development of Ghidrion significantly, as the information needed for each step had to be gathered from multiple sources, and certain parts had to be implemented multiple times as a result of changing and improving understanding of how Ghidra's API works.

Based on this, the additional information provided in this thesis, and in particular in Sections 5.1 and 5.2, will prove valuable to developers creating similar plugins in the future. As mentioned in Section 6.1, the source code of Ghidrion itself, along with the JavaDoc comments for classes and methods, provides more detailed documentation showing the specific application of the principles outlined in this thesis.

7.2 Improvements in the Workflow of an Analyst

In this section, the authors discuss the main advantages of using Ghidrion over either Morion alone or a combination of Morion and Ghidra.

7.2.1 Create Init YAML

Ghidrion's support for adding function hooks is the most fundamental improvement in an analyst's workflow. It allows them to automatically add hooks for all invocations of external functions (i.e. functions from external libraries), which would

otherwise have to be added manually on an individual basis. It further removes the need to manually type addresses, thus eliminating a source of error. The only data that still needs to be entered is the library name, the reason for which is discussed in Section 7.3.2.

When adding memory entries and registers to the Init YAML, Ghidrion mainly provides convenience functionality, like the ability to split large memory values across addresses, as discussed in Section 6.2.2. It also provides some input validation by enforcing maximum value lengths and that they contain only valid hexadecimal characters.

7.2.2 Tracing

Since Ghidra already had a way to access a GDB environment in its debugger tool, the authors did not re-implement this in Ghidrion.

7.2.3 Analyse Traced YAML

Since the Traced YAML does not list the entry and leave states side by side, it can be rather cumbersome to compare the values. Ghidrion simplifies this by juxtaposing the values and indicating which elements changed during the executed instructions.

It also highlights the executed instructions in Ghidra's Listing window, allowing an analyst to quickly see which parts of the binary were executed. The main advantage over just looking at the Traced YAML is that the corresponding instructions in the C pseudocode are also marked in the Decompile window.

7.3 Future Work

During the development of Ghidrion, the authors collected ideas that did not fit the scope of this thesis. This section provides a list of such ideas, with the reasons why they are too complex to be considered for implementation as part of this work. It further discusses known limitations of the current implementation.

7.3.1 Hooking Functions in Sections Other Than `.text`

For ELF binaries, Ghidrion detects external functions outside of `.text`. This is not a problem when interacting with Ghidrion, but when exporting the generated Init YAML, it is considerably more difficult to calculate a valid return address, especially in the `.plt` section, since it is not possible to just use the next instruction. Hooking function calls in these sections is desirable because all calls to any external function will eventually execute the instructions in these sections, which would make it possible to hook any execution of a particular function with a single entry in the YAML. For now, analysts can easily ignore entries outside of `.text` by using a text filter. This is demonstrated in Figure 6.1.

7.3.2 Automatic Library Detection

Currently, the analyst has to provide the library name of the function they wish to hook. Ideally, instead of just a text field in the panel where hooks can be added (as shown in Figure 6.1), the library name would be another value by which hooks

can be filtered, analogous to the function and block name and function address. However, this could not be implemented for two reasons. First, the library name would have to be the same (i.e. the same spelling, capitalisation, presence or absence of version number, etc.) as what is used in Morion. This would be a feasible change but was not attempted because of the second problem.

Fundamentally, the library name has to be stored somewhere in the binary, since the loader needs to know which dynamic library to link function calls to. However, based on checking the `Function`, `Symbol`, and `Reference` objects along with the use of `FunctionManager`, `SymbolTable`, `ReferenceManager`, `Memory`, and `ExternalManager`, it seems that Ghidra's binary loader does not load this information into the respective objects if the external library is not present on the system running Ghidra. And since the machine running the binary and the machine running Ghidra are often not the same, the authors cannot assume that the library is present. Listing 7.1 shows the error in the import logs when loading the example binary.

```

1  ----- Loading /Users/valentinhuber/Downloads/strlen -----
8  Linking external programs to strlen...
9    [libc.so.6] -> not found
10 ----- [strlen] Resolve 7 external symbols -----
11 Unresolved external symbols which remain: 7

```

LISTING 7.1: Excerpt from Logs when Importing Example Code Showing Error Concerning External Symbols.

Looking at how other people have attempted to solve this issue, it seems that the most common approach is to retrieve the path of the library and then manually import it [72], [73], [74], [75]. For the reason explained above, this does not work and, as expected, the function calls to retrieve the path (see Listing 7.2) return `null`.

```

6  externalLocation = getFunctionAt(toAddr("004341fe")) |
   → .getThunkedFunction(True).getExternalLocation()
12 libPath = currentProgram.getExternalManager() |
   → .getExternalLibrary(externalLocation.getLibraryName()) |
   → .getAssociatedProgramPath()

```

LISTING 7.2: Excerpt of Code Proposed in [72] To Load External Symbols Returning `null` in the Author's Java Implementation. See Appendix 3 for the Full Proposal.

The authors have found no other way to retrieve this data using Ghidra's current capabilities. Implementing this feature would seem to require either modifying Ghidra's binary loader or writing custom code to parse binaries and read the library name (see Section 5.1.3). Both options are beyond the scope of this project and were not pursued.

7.3.3 Automatic Register Detection

As suggested in Section 4.2.1, for future error prevention, Ghidra might give analysts a list of available registers to choose from, to limit the exported values to valid register names. This raises two issues: First, as mentioned in Section 7.3.2, the names used by Morion need to match the names used in Ghidra, which may require either

changes in Morion or a translation layer in Ghidrion, defeating the point of it being automatic.

However, the main problem in implementing this is that while Ghidra provides a list of registers for a given binary (see Section 5.1.4), this list contains many irrelevant registers. Appendix 4 provides a table with attributes for each register. These could be used to filter the list to show only relevant registers, but the authors did not find a way to make this filter narrow enough and therefore opted to use a text field to prompt for the register name.

7.3.4 Further Ideas

The following are additional ideas that do not warrant their own section:

- Hooked functions should be marked in Ghidra's Listing window.
- Ghidrion should have the ability to open the GDB environment in Ghidra's Debugger tool and preload Morion's tracing script. This would require Ghidrion to know Morion's installation path.
- Ghidrion should be automatically built and released on a feature branch merge (known as continuous delivery). Currently, releases have to be manually created using GhidraDev (see Section 5.1.1) and published.

Indices

1 Bibliography

- [1] “Cyber-defence (cyd) campus.” (n.d.), [Online]. Available: <https://cydcampus.ch> (visited on May 30, 2023).
- [2] “Ghidra.” (n.d.), [Online]. Available: <https://ghidra-sre.org/> (visited on Apr. 10, 2023).
- [3] D. Andriessse, *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. no starch press, 2018.
- [4] J. Erickson, *Hacking: The Art of Exploitation*. no starch press, 2008.
- [5] T. Cipresso and M. Stamp, “Software reverse engineering,” in *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 659–696, ISBN: 978-3-642-04117-4. DOI: [10.1007/978-3-642-04117-4_31](https://doi.org/10.1007/978-3-642-04117-4_31). [Online]. Available: https://doi.org/10.1007/978-3-642-04117-4_31.
- [6] G. Canfora, M. Di Penta, and L. Cerulo, “Achievements and challenges in software reverse engineering,” *Commun. ACM*, vol. 54, no. 4, pp. 142–151, Apr. 2011, ISSN: 0001-0782. DOI: [10.1145/1924421.1924451](https://doi.org/10.1145/1924421.1924451). [Online]. Available: <https://doi.org/10.1145/1924421.1924451>.
- [7] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.
- [8] S. Alrabae, M. Debbabi, P. Shirani, *et al.*, “Binary analysis overview,” in *Binary Code Fingerprinting for Cybersecurity: Application to Malicious Code Fingerprinting*. Cham: Springer International Publishing, 2020, pp. 7–44, ISBN: 978-3-030-34238-8. DOI: [10.1007/978-3-030-34238-8_2](https://doi.org/10.1007/978-3-030-34238-8_2). [Online]. Available: https://doi.org/10.1007/978-3-030-34238-8_2.
- [9] N. Nethercote, “Dynamic binary analysis and instrumentation,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-606, Nov. 2004. DOI: [10.48456/tr-606](https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf). [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf>.
- [10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, “Selective symbolic execution,” 2009. [Online]. Available: <http://infoscience.epfl.ch/record/139393>.
- [11] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: [10.1145/3182657](https://doi.org/10.1145/3182657). [Online]. Available: <https://doi.org/10.1145/3182657>.
- [12] “Z3.” (n.d.), [Online]. Available: <https://github.com/Z3Prover/z3> (visited on May 30, 2023).
- [13] J. Salwan. “Triton — improve the taint analysis.” (May 16, 2020), [Online]. Available: <https://github.com/JonathanSalwan/Triton/issues/908#issuecomment-629683535> (visited on Jun. 7, 2023).

- [14] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy*, 2016.
- [15] F. Saudel and J. Salwan, “Triton: A dynamic symbolic execution framework,” in *Symposium sur la sécurité des technologies de l’information et des communications*, ser. SSTIC, Rennes, France, Jun. 2015, pp. 31–54.
- [16] M. Mossberg, F. Manzano, E. Hennenfent, *et al.*, *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*, Nov. 2019. DOI: 10.1109/ASE.2019.00133. [Online]. Available: <https://github.com/trailofbits/manticore>.
- [17] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *SIGPLAN Not.*, vol. 46, no. 3, pp. 265–278, Mar. 2011, ISSN: 0362-1340. DOI: 10.1145/1961296.1950396. [Online]. Available: <https://doi.org/10.1145/1961296.1950396>.
- [18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.
- [19] R. David, S. Bardin, T. D. Ta, *et al.*, “BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, IEEE Computer Society, 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43. [Online]. Available: <https://doi.org/10.1109/SANER.2016.43>.
- [20] E. Cheng, “Binary analysis and symbolic execution with angr,” Ph.D. dissertation, PhD thesis, The MITRE Corporation, 2016.
- [21] “Triton: A dynamic binary analysis library.” (n.d.), [Online]. Available: <https://triton-library.github.io/> (visited on Mar. 28, 2023).
- [22] “Libtriton: Triton.” (n.d.), [Online]. Available: <https://triton-library.github.io/documentation/doxygen/index.html> (visited on May 30, 2023).
- [23] A. Niemetz and M. Preiner, “Bitwuzla at the SMT-COMP 2020,” *CoRR*, vol. abs/2006.01621, 2020. arXiv: 2006.01621. [Online]. Available: <https://arxiv.org/abs/2006.01621>.
- [24] “Tritondse.” (n.d.), [Online]. Available: <https://github.com/quarkslab/tritondse> (visited on May 30, 2023).
- [25] “Ida pro.” (n.d.), [Online]. Available: <https://hex-rays.com/ida-pro/> (visited on Apr. 10, 2023).
- [26] “Binary ninja.” (n.d.), [Online]. Available: <https://binary.ninja/> (visited on Apr. 10, 2023).
- [27] A. Novoseltseva. “Angryghidra.” (n.d.), [Online]. Available: <https://github.com/Nalen98/AngryGhidra> (visited on Apr. 10, 2023).
- [28] “Summ3r of h4ck 2020. results of the program.” (n.d.), [Online]. Available: <https://prog.world/summ3r-of-h4ck-2020-results-of-the-program/> (visited on Apr. 10, 2023).
- [29] L. Borzacchiello, E. Coppa, and C. Demetrescu, “Seninja: A symbolic execution plugin for binary ninja,” *SoftwareX*, vol. 20, p. 101219, 2022.
- [30] A. Skliarova, *The taming of gorynych 2, or symbolic performance in ghidra*, n.d. [Online]. Available: https://habr-com.translate.google.ru/company/dsec/blog/520206/?_x_tr_sl=ru&_x_tr_tl=en&_x_tr_hl=en&_x_tr_pto=sc&_x_tr_hist=true (visited on Apr. 10, 2023).

- [31] “Cle - cle documentation.” (n.d.), [Online]. Available: <https://docs.angr.io/projects/cle/en/latest/quickstart.html#finding-shared-libraries> (visited on Apr. 10, 2023).
- [32] J. Ziegler, “Edge of the art in vulnerability research version 4 of 4,” Two Six Labs, Tech. Rep., 2021.
- [33] “Angr management.” (n.d.), [Online]. Available: <https://github.com/angr/angr-management> (visited on Mar. 23, 2023).
- [34] “Scripting angr management.” (n.d.), [Online]. Available: https://docs.angr.io/en/latest/extending-angr/angr_management.html (visited on Mar. 23, 2023).
- [35] “Ponce.” (n.d.), [Online]. Available: <https://github.com/illera88/Ponce> (visited on Mar. 28, 2023).
- [36] “Introduction - ponce.” (n.d.), [Online]. Available: <https://docs.idaponce.com/> (visited on Mar. 28, 2023).
- [37] “Manticore ui.” (n.d.), [Online]. Available: <https://github.com/trailofbits/ManticoreUI> (visited on Mar. 28, 2023).
- [38] M. Mossberg, F. Manzano, E. Hennenfent, *et al.*, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189. DOI: [10.1109/ASE.2019.00133](https://doi.org/10.1109/ASE.2019.00133).
- [39] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, vol. 3, no. 37, pp. 2–1, 2014.
- [40] “Webassembly.” (n.d.), [Online]. Available: <https://webassembly.org/> (visited on Jun. 2, 2023).
- [41] “Manticore.” (n.d.), [Online]. Available: <https://github.com/trailofbits/manticore> (visited on Mar. 28, 2023).
- [42] A. Chang. “Mui: Visualizing symbolic execution with manticore and binary ninja.” (Nov. 2021), [Online]. Available: <https://blog.trailofbits.com/2021/11/17/mui-visualizing-symbolic-execution-with-manticore-and-binary-ninja/> (visited on Mar. 28, 2023).
- [43] “Welcome to manticore’s documentation!” (n.d.), [Online]. Available: <https://manticore.readthedocs.io/en/latest/index.html> (visited on Jun. 2, 2023).
- [44] D. Pfammatter, personal communication, May 2023.
- [45] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE ’81, San Diego, California, USA: IEEE Press, 1981, pp. 439–449, ISBN: 0897911466.
- [46] “Jython: Python for the java platform.” (n.d.), [Online]. Available: <https://github.com/jython/jython> (visited on Jun. 1, 2023).
- [47] “Ghidra api — ghidrascript.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/script/GhidraScript.html (visited on Jun. 1, 2023).
- [48] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. no starch press, 2020.
- [49] “Ghidra api — flatprogramapi.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/flatapi/FlatProgramAPI.html (visited on Jun. 1, 2023).
- [50] “Ghidra api — program.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Program.html (visited on May 24, 2023).

- [51] “Ghidra api — abstractanalyzer.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/services/AbstractAnalyzer.html (visited on Jun. 1, 2023).
- [52] “Ghidra api — programplugin.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/plugin/ProgramPlugin.html (visited on May 24, 2023).
- [53] A. David, *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*. Packt Publishing Ltd., 2020.
- [54] “Ghidra api — pluginevent.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/framework/plugintool/PluginEvent.html (visited on Jun. 1, 2023).
- [55] “Ghidra api — componentprovider.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/docking/ComponentProvider.html (visited on May 24, 2023).
- [56] “Ghidra api — plugin.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/framework/plugintool/Plugin.html (visited on Jun. 1, 2023).
- [57] “Ghidra api — abstractprogramloader.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/util/opinion/AbstractProgramLoader.html (visited on Jun. 1, 2023).
- [58] “Ghidra api — gfilesystem.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/formats/gfilesystem/GFileSystem.html (visited on Jun. 1, 2023).
- [59] “Ghidra api — exporter.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/util/exporter/Exporter.html (visited on Jun. 1, 2023).
- [60] “Ghidra api — overview.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/index.html (visited on May 24, 2023).
- [61] “Ghidra api — functionmanager.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/FunctionManager.html (visited on May 24, 2023).
- [62] “Ghidra api — referencemanager.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/model/symbol/ReferenceManager.html (visited on May 24, 2023).
- [63] “Ghidra api — memory.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/model/mem/Memory.html (visited on May 24, 2023).
- [64] “Ghidra api — programcontext.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/ProgramContext.html (visited on May 24, 2023).
- [65] “Ghidra api — listingcontextaction.” (n.d.), [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/context/ListingContextAction.html (visited on May 24, 2023).
- [66] S. Burbeck, *Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc)*, 1987. [Online]. Available: http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf (visited on Jun. 1, 2023).
- [67] A. Fowler. “A swing architecture overview.” (n.d.), [Online]. Available: <https://www.oracle.com/java/technologies/a-swing-architecture.html> (visited on May 24, 2023).

- [68] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.
- [69] "Ghidrathon." (n.d.), [Online]. Available: <https://github.com/mandiant/Ghidrathon> (visited on Jun. 1, 2023).
- [70] "Pyhidra." (n.d.), [Online]. Available: <https://github.com/dod-cyber-crime-center/pyhidra> (visited on Jun. 1, 2023).
- [71] "Ghidradev readme." (n.d.), [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraBuild/EclipsePlugins/GhidraDev/GhidraDevPlugin/GhidraDev_README.html (visited on Jun. 6, 2023).
- [72] "Accessing external function address space." (Feb. 19, 2022), [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra/issues/1882#issuecomment-1046047575> (visited on Jun. 1, 2023).
- [73] A. Strelsky. "Retrieving external location from symbol address." (May 3, 2020), [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra/issues/1833> (visited on Jun. 1, 2023).
- [74] "Idiomatic way of creating external data." (May 7, 2019), [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra/issues/578> (visited on Jun. 1, 2023).
- [75] A. Osti. "How to solve references between multiple raw binary images?" (Nov. 12, 2019), [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra/issues/1236> (visited on Jun. 1, 2023).

2 List of Figures

2.1	Symbolic Execution Example: Evolution of the Symbolic State	8
2.2	Symbolic Execution Design Dimensions	9
3.1	AngryGhidra: Set Blank State Address	17
3.2	AngryGhidra: Find Address and Avoid Addresses	18
3.3	AngryGhidra: Filled Panel Reporting a Solution	19
4.1	Morion’s Workflow	21
6.1	Ghidrion: Adding Hooks	34
6.2	Ghidrion: Adding Hooks Using the Context Menu	35
6.3	Ghidrion: Changing Hooks Using the Context Menu	35
6.4	Ghidrion: Adding Memory Entries	36
6.5	Ghidrion: Adding Registers	37
6.6	Ghidrion: Diff View for Memory Entries	39
6.7	Ghidrion: Diff View for Registers	40
6.8	Ghidrion: Marking Executed Instructions	40

3 List of Tables

A1	Registers Identified by Ghidra	56
----	--	----

4 List of Listings

1.1	Example Code — <code>strlen.c</code>	3
2.1	Pseudocode Including a Nested <code>if</code> Statement	7
3.1	Main Method of <code>strlen.c</code>	16
4.1	Example Init YAML File	24
4.2	Excerpt From the Instructions of the Example Traced YAML File	25
4.3	Excerpt From the Entry and Leave State of the Example Traced YAML File	26
5.1	Ghidrion: Gathering Hookable Functions	29
5.2	Ghidrion: Creating Context Menu	31
7.1	Excerpt from Logs when Importing Example Code	43
7.2	Code Excerpt To Load External Symbols	43
A1	Example Traced YAML File	54
A2	Code To Load External Symbols	55

Appendix

1 Installation of AngryGhidra

The installation of AngryGhidra is documented in its GitHub README [27]. It is quite straight-forward:

1. Install Python3 and make sure it is added to your PATH environment variable.
2. In a terminal, run `python3 -m pip install angr` to install angr.
3. Download and run the latest release of Ghidra.
4. Download AngryGhidra from [GitHub Releases](#). It has to be the zip file that matches the version of Ghidra.
5. In Ghidra, install the extension by `File → Install Extensions...` and choosing the AngryGhidra zip file.

2 Full Example Traced YAML File

```

1 hooks:
2   libc:
3     printf:
4       - entry: '0x4005b4'
5         leave: '0x4005b8'
6         mode: skip
7       - entry: '0x4005d0'
8         leave: '0x4005d4'
9         mode: skip
10    strlen:
11      - entry: '0x400594'
12        leave: '0x400598'
13        mode: model
14  info:
15    arch: armv7
16    thumb: false
17  instructions:
18    - - '0x00400588'
19      - 00 30 a0 e1
20      - 'mov r3, r0'
21      - 'L15: `s = (char *) calloc(BUF_LENGTH, sizeof(char));`'
22      - - '0x0040058c'
23        - 08 30 0b e5
24        - 'str r3, [fp, #-8]'
25        - 'L15: `s = (char *) calloc(BUF_LENGTH, sizeof(char));`'
26      - - '0x00400590'
27        - 08 00 1b e5
28        - 'ldr r0, [fp, #-8]'
29        - 'L18: `if(strlen(s) == 2) {'`
30      - - '0x00400594'
31        - 59 ff ef ea
32        - 'b #-0x400294'
33        - '// Hook: libc:strlen (on=entry, mode=model)'
34      - - '0x00000300'
35        - 07 00 a0 e3
36        - 'mov r0, #0x7'
37        - '// Hook: libc:strlen (on=leave, mode=model)'
38      - - '0x00000304'
39        - 00 00 40 e3
40        - 'movt r0, #0x0'
41        - '// Hook: libc:strlen (on=leave, mode=model)'
42      - - '0x00000308'
43        - a2 00 10 ea
44        - 'b #0x400290'
45        - '// Hook: libc:strlen (on=leave, mode=model)'
46      - - '0x00400598'
47        - 00 30 a0 e1
48        - 'mov r3, r0'
49        - 'L18: `if(strlen(s) == 2) {'`
50      - - '0x0040059c'
51        - 02 00 53 e3
52        - 'cmp r3, #2'
53        - 'L18: `if(strlen(s) == 2) {'`
54      - - '0x004005a0'
55        - 06 00 00 1a
56        - 'bne #0x4005c0'
57        - 'L18: `if(strlen(s) == 2) {'`
58      - - '0x004005c0'
59        - 08 10 1b e5
60      - 'ldr r1, [fp, #-8]'

```

```

61     - 'L22: `printf("strlen('%s') != 2\n", s);`'
62     - '0x004005c4'
63     - 1c 30 9f e5
64     - 'ldr r3, [pc, #0x1c]'
65     - 'L22: `printf("strlen('%s') != 2\n", s);`'
66     - '0x004005c8'
67     - 03 30 8f e0
68     - 'add r3, pc, r3'
69     - 'L22: `printf("strlen('%s') != 2\n", s);`'
70     - '0x004005cc'
71     - 03 00 a0 e1
72     - 'mov r0, r3'
73     - 'L22: `printf("strlen('%s') != 2\n", s);`'
74     - '0x004005d0'
75     - 0a ff ef ea
76     - 'b #-0x4003d0'
77     - '// Hook: libc:printf (on=entry, mode=skip)'
78     - '0x00000200'
79     - 17 00 a0 e3
80     - 'mov r0, #0x17'
81     - '// Hook: libc:printf (on=leave, mode=skip)'
82     - '0x00000204'
83     - 00 00 40 e3
84     - 'movt r0, #0x0'
85     - '// Hook: libc:printf (on=leave, mode=skip)'
86     - '0x00000208'
87     - f1 00 10 ea
88     - 'b #0x4003cc'
89     - '// Hook: libc:printf (on=leave, mode=skip)'
90     - '0x004005d4'
91     - 00 30 a0 e3
92     - 'mov r3, #0'
93     - 'L23: `return EXIT_SUCCESS;`'
94     - '0x004005d8'
95     - 03 00 a0 e1
96     - 'mov r0, r3'
97     - 'L24: `}`'
98     - '0x004005dc'
99     - 04 d0 4b e2
100    - 'sub sp, fp, #4'
101    - 'L24: `}`'
102    - '0x004005e0'
103    - 00 88 bd e8
104    - 'pop {fp, pc}'
105    - 'L24: `}`'
106
107    states:
108    entry:
109    addr: '0x00400588'
110    mems:
111    '0x004005e8': ['0x10']
112    '0x004005e9': ['0x01']
113    '0x004005ea': ['0x00']
114    '0x004005eb': ['0x00']
115    '0x00412190': ['0x41']
116    '0x00412191': ['0x42', $$]
117    '0x00412192': ['0x43', $$]
118    '0x00412193': ['0x44']
119    '0x00412194': ['0x45', $$]
120    '0x00412195': ['0x46']
121    '0x00412196': ['0x47']
122    '0x00412197': ['0x00']
123    '0xbefffb6c': ['0x00']

```

```
124     '0xbeffffb6d' : ['0x00']
125     '0xbeffffb6e' : ['0x00']
126     '0xbeffffb6f' : ['0x00']
127     '0xbeffffb70' : ['0x00']
128     '0xbeffffb71' : ['0x00']
129     '0xbeffffb72' : ['0x00']
130     '0xbeffffb73' : ['0x00']
131     '0xbeffffb74' : ['0xa1']
132     '0xbeffffb75' : ['0xd5']
133     '0xbeffffb76' : ['0xee']
134     '0xbeffffb77' : ['0xb6']
135     regs:
136     r0: ['0x00412190']
137     r1: ['0x00412190']
138     r11: ['0xbeffffb74']
139     r3: ['0x00412198']
140     sp: ['0xbeffffb60']
141     z: ['0x00000000']
142     leave:
143     addr: '0xb6eed5a0'
144     mems:
145     '0x004005e8' : ['0x10']
146     '0x004005e9' : ['0x01']
147     '0x004005ea' : ['0x00']
148     '0x004005eb' : ['0x00']
149     '0x00412190' : ['0x41']
150     '0x00412191' : ['0x42']
151     '0x00412192' : ['0x43']
152     '0x00412193' : ['0x44']
153     '0x00412194' : ['0x45']
154     '0x00412195' : ['0x46']
155     '0x00412196' : ['0x47']
156     '0x00412197' : ['0x00']
157     '0xbeffffb6c' : ['0x90']
158     '0xbeffffb6d' : ['0x21']
159     '0xbeffffb6e' : ['0x41']
160     '0xbeffffb6f' : ['0x00']
161     '0xbeffffb70' : ['0x00']
162     '0xbeffffb71' : ['0x00']
163     '0xbeffffb72' : ['0x00']
164     '0xbeffffb73' : ['0x00']
165     '0xbeffffb74' : ['0xa1']
166     '0xbeffffb75' : ['0xd5']
167     '0xbeffffb76' : ['0xee']
168     '0xbeffffb77' : ['0xb6']
169     regs:
170     r0: ['0x00000000']
171     r1: ['0x00000000']
172     r11: ['0x00000000']
173     r3: ['0x00000000']
174     sp: ['0xbeffffb78']
175     z: ['0x00000001']
```

LISTING A1: Example Traced YAML File

3 Full Code Proposed to Load External Symbols

```
1 from ghidra.app.util import SymbolPath
2 from ghidra.app.util import NamespaceUtils
3
4 # Have to sub in your own address
5 # The address should be the address of the Thunk
6 externalLocation = getFunctionAt(toAddr("004341fe"))
7     → .getThunkedFunction(True).getExternalLocation()
8 externalSymbol = externalLocation.getSymbol()
9 program = externalSymbol.getProgram()
10
11 # Need to open the external program
12 # Adapted from https://github.com/saruman9/ghidra_scripts/blob/master
13     → /FindExternalReferences.java
14 libPath = currentProgram.getExternalManager()
15     → .getExternalLibrary(externalLocation.getLibraryName())
16     → .getAssociatedProgramPath()
17 projectData = state.getProject().getProjectData()
18 libFile = projectData.getFile(libPath)
19 externalProgram = libFile.getImmutableDomainObject(libFile,
20     → DomainFile.DEFAULT_VERSION, monitor)
21
22 # Adapted from https://github.com/NationalSecurityAgency/ghidra/blob
23     → /da94eb86bd2b89c8b0ab9bd89e9f0dc5a3157055/Ghidra/Features/Base/src
24     → /main/java/ghidra/app/plugin/core/gotoquery/GoToHelper.java)
25 label = externalLocation.getOriginalImportedName()
26 symbolPath = SymbolPath(label)
27
28 for x in NamespaceUtils.getSymbols(symbolPath, externalProgram):
29     print(x)
30     if x.isExternalEntryPoint():
31         sym = x
32
33 loc = sym.getProgramLocation()
34 print(loc.getAddress())
35 fn = prog.getFunctionManager().getFunctionAt(loc.getAddress())
```

LISTING A2: Code Proposed in [72] To Load External Symbols

4 Registers Identified by Ghidra

TABLE A1: Registers Identified by Ghidra in Its ProgramContext [64] With Corresponding Non-Redundant Information

Name	Group	ParentRegister	TypeFlags	BitLength	ChildRegister Count	isBaseRegister	isProgramCounter	isVectorRegister
r0			0x00	0x20	0	x		
r1			0x00	0x20	0	x		
r2			0x00	0x20	0	x		
r3			0x00	0x20	0	x		
r4			0x00	0x20	0	x		
r5			0x00	0x20	0	x		
r6			0x00	0x20	0	x		
r7			0x00	0x20	0	x		
r8			0x00	0x20	0	x		
r9			0x00	0x20	0	x		
r10			0x00	0x20	0	x		
r11			0x00	0x20	0	x		
r12			0x00	0x20	0	x		
sp			0x00	0x20	0	x		
lr			0x00	0x20	0	x		
pc			0x04	0x20	0	x	x	
NG			0x00	0x08	0	x		
ZR			0x00	0x08	0	x		
CY			0x00	0x08	0	x		
OV			0x00	0x08	0	x		
tmpNG			0x00	0x08	0	x		
tmpZR			0x00	0x08	0	x		
tmpCY			0x00	0x08	0	x		
tmpOV			0x00	0x08	0	x		
shift_carry			0x00	0x08	0	x		
TB			0x00	0x08	0	x		
Q			0x00	0x08	0	x		
GE1			0x00	0x08	0	x		
GE2			0x00	0x08	0	x		
GE3			0x00	0x08	0	x		
GE4			0x00	0x08	0	x		
cpsr			0x00	0x20	0	x		
spsr			0x00	0x20	0	x		
mult_addr			0x00	0x20	0	x		
r14_svc			0x00	0x20	0	x		
r13_svc			0x00	0x20	0	x		
spsr_svc			0x00	0x20	0	x		
mult_dat8		mult_dat16	0x00	0x40	0			

Name	Group	ParentRegister	TypeFlags	BitLength	ChildRegister Count	isBaseRegister	isProgramCounter	isVectorRegister
mult_dat16			0x00	0x80	1	x		
fpsr			0x00	0x20	0	x		
ISAModeSwitch		fpsid	0x00	0x08	0			
fpsid		fpsid	0x00	0x20	1	x		
fpscr			0x00	0x20	0	x		
fpexc			0x00	0x20	0	x		
mvfr0			0x00	0x20	0	x		
mvfr1			0x00	0x20	0	x		
fp0			0x00	0x50	0	x		
fp1			0x00	0x50	0	x		
fp2			0x00	0x50	0	x		
fp3			0x00	0x50	0	x		
fp4			0x00	0x50	0	x		
fp5			0x00	0x50	0	x		
fp6			0x00	0x50	0	x		
fp7			0x00	0x50	0	x		
cr0			0x00	0x20	0	x		
cr1			0x00	0x20	0	x		
cr2			0x00	0x20	0	x		
cr3			0x00	0x20	0	x		
cr4			0x00	0x20	0	x		
cr5			0x00	0x20	0	x		
cr6			0x00	0x20	0	x		
cr7			0x00	0x20	0	x		
cr8			0x00	0x20	0	x		
cr9			0x00	0x20	0	x		
cr10			0x00	0x20	0	x		
cr11			0x00	0x20	0	x		
cr12			0x00	0x20	0	x		
cr13			0x00	0x20	0	x		
cr14			0x00	0x20	0	x		
cr15			0x00	0x20	0	x		
s0		d0	0x00	0x20	0			
s1		d0	0x00	0x20	0			
s2		d1	0x00	0x20	0			
s3		d1	0x00	0x20	0			
s4		d2	0x00	0x20	0			
s5		d2	0x00	0x20	0			
s6		d3	0x00	0x20	0			
s7		d3	0x00	0x20	0			
s8		d4	0x00	0x20	0			
s9		d4	0x00	0x20	0			

Name	Group	ParentRegister	TypeFlags	BitLength	ChildRegister Count	isBaseRegister	isProgramCounter	isVectorRegister
s10		d5	0x00	0x20	0			
s11		d5	0x00	0x20	0			
s12		d6	0x00	0x20	0			
s13		d6	0x00	0x20	0			
s14		d7	0x00	0x20	0			
s15		d7	0x00	0x20	0			
s16		d8	0x00	0x20	0			
s17		d8	0x00	0x20	0			
s18		d9	0x00	0x20	0			
s19		d9	0x00	0x20	0			
s20		d10	0x00	0x20	0			
s21		d10	0x00	0x20	0			
s22		d11	0x00	0x20	0			
s23		d11	0x00	0x20	0			
s24		d12	0x00	0x20	0			
s25		d12	0x00	0x20	0			
s26		d13	0x00	0x20	0			
s27		d13	0x00	0x20	0			
s28		d14	0x00	0x20	0			
s29		d14	0x00	0x20	0			
s30		d15	0x00	0x20	0			
s31		d15	0x00	0x20	0			
d0		q0	0x00	0x40	2			
d1		q0	0x00	0x40	2			
d2		q1	0x00	0x40	2			
d3		q1	0x00	0x40	2			
d4		q2	0x00	0x40	2			
d5		q2	0x00	0x40	2			
d6		q3	0x00	0x40	2			
d7		q3	0x00	0x40	2			
d8		q4	0x00	0x40	2			
d9		q4	0x00	0x40	2			
d10		q5	0x00	0x40	2			
d11		q5	0x00	0x40	2			
d12		q6	0x00	0x40	2			
d13		q6	0x00	0x40	2			
d14		q7	0x00	0x40	2			
d15		q7	0x00	0x40	2			
d16		q8	0x00	0x40	0			
d17		q8	0x00	0x40	0			
d18		q9	0x00	0x40	0			
d19		q9	0x00	0x40	0			

Name	Group	ParentRegister	TypeFlags	BitLength	ChildRegister Count	isBaseRegister	isProgramCounter	isVectorRegister
d20		q10	0x00	0x40	0			
d21		q10	0x00	0x40	0			
d22		q11	0x00	0x40	0			
d23		q11	0x00	0x40	0			
d24		q12	0x00	0x40	0			
d25		q12	0x00	0x40	0			
d26		q13	0x00	0x40	0			
d27		q13	0x00	0x40	0			
d28		q14	0x00	0x40	0			
d29		q14	0x00	0x40	0			
d30		q15	0x00	0x40	0			
d31		q15	0x00	0x40	0			
q0	NEON		0x80	0x80	2	x		x
q1	NEON		0x80	0x80	2	x		x
q2	NEON		0x80	0x80	2	x		x
q3	NEON		0x80	0x80	2	x		x
q4	NEON		0x80	0x80	2	x		x
q5	NEON		0x80	0x80	2	x		x
q6	NEON		0x80	0x80	2	x		x
q7	NEON		0x80	0x80	2	x		x
q8	NEON		0x80	0x80	2	x		x
q9	NEON		0x80	0x80	2	x		x
q10	NEON		0x80	0x80	2	x		x
q11	NEON		0x80	0x80	2	x		x
q12	NEON		0x80	0x80	2	x		x
q13	NEON		0x80	0x80	2	x		x
q14	NEON		0x80	0x80	2	x		x
q15	NEON		0x80	0x80	2	x		x
contextreg			0x08	0x40	19	x		
TMode		contextreg	0x08	0x01	0			
LRset		contextreg	0x48	0x01	0			
RETOVERRIDE		contextreg	0x48	0x01	0			
CALLOVERRIDE		contextreg	0x48	0x01	0			
TEEMode		contextreg	0x08	0x01	0			
condit		contextreg	0x48	0x09	0			
itmode		contextreg	0x08	0x01	0			
cond_full		contextreg	0x08	0x04	0			
cond_base		contextreg	0x08	0x03	0			
cond_true		contextreg	0x08	0x01	0			
cond_shft		contextreg	0x08	0x05	0			
cond_mask		contextreg	0x08	0x04	0			
counter		contextreg	0x08	0x05	0			

Name	Group	ParentRegister	TypeFlags	BitLength	ChildRegister Count	isBaseRegister	isProgramCounter	isVectorRegister
regNum		contextreg	0x08	0x05	0			
counter2		contextreg	0x08	0x03	0			
reg2Num		contextreg	0x08	0x05	0			
regInc		contextreg	0x08	0x02	0			
ARMcond		contextreg	0x08	0x01	0			
ARMcondCk		contextreg	0x08	0x01	0			

5 Initial Thesis Description

5.1 Titel

Ghidra-Erweiterung für das Morion Symbolic Execution Tool

5.2 Beschreibung

Im Rahmen seines Forschungsprogrammes im Bereich Vulnerability Research, erprobt der Cyber-Defense Campus von armasuisse W+T die Anwendbarkeit von Symbolic Execution auf echte, sogenannte real-world, Binaries. Um an den momentan existierenden praktischen Limitationen zu forschen, wird an der Entwicklung eines Proof-of-Concept (PoC) Tool namens Morion gearbeitet, welches im Rahmen dieses Projektes erweitert werden soll. Morion bietet die Funktionalität, verschiedene auf Symbolic Execution basierende Analysen auf selektiven Pfaden innerhalb eines echten ARMv7 Binaries auszuführen.

Im Rahmen dieses Projektes sollen folgende Aufgabestellungen bearbeitet werden:

Morion Ghidra-Erweiterung (Hauptaufgabe)

- Durchführen einer Analyse existierender Symbolic Execution Erweiterung für populäre Reverse Engineering Frameworks wie Ghidra, IDA Pro oder Binary Ninja (Literaturstudie)
- Einrichten einer Test-/Entwicklungsumgebung (Ghidra, Morion, QEMU VM)
- Einarbeitung in die Kernfunktionalitäten von Morion, den grundlegenden Konzepten von Reverse Engineering und (Binary) Symbolic Execution, sowie der zu verwendenden Tools
- Ausarbeitung und Entwicklung einer Erweiterung für das Reverse Engineering Framework Ghidra, welches als Frontend für Morion agieren soll
- Verwendung der GDB-Debugging Integration in Ghidra um mit Hilfe von Morion konkrete Execution Traces von Ziel-Binaries zu sammeln (Tracing)
- Verwendung der Traces zur Ausführung verschiedener von Morion implementierter Symbolic Execution Analysen, sowie Integration/Darstellung relevanter Resultate in Ghidra Morion Pfad-Analyse Modul (Optionale Aufgabe)
- Ausarbeitung einer auf Symbolic Execution basierenden Analyse, welche eine mögliche Lösung zum Erreichen einer Ziel-Adresse, ausgehend von einer vordefinierten Start-Adresse, zurückgibt
- Möglichkeit kontrollierbare Bereiche als symbolisch zu markieren, basierend auf welchen das Analyse-Modul eine mögliche Lösung berechnet
- Verwendung von Ghidra zum Erstellen sogenannter Control Flow Graphs (CFGs), welche zum Auffinden möglicher Pfade dienen

Die Abgabe der Projekt-Resultate soll anhand des entwickelten Source Codes erfolgen, sowie in Form eines zugehörigen technischen Berichtes, welcher die Erarbeiteten Erkenntnisse und Dokumentation zusammenfasst.

Referenzen

- "Triton: A Dynamic Binary Analysis Library", <https://triton-library.github.io/>
- "Ghidra: Software Reverse Engineering Toolsuite", <https://ghidra-sre.org/>
- "Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly", D. Andriessse, 2019
- "Ponce: IDA Pro Plugin based on Triton", <https://github.com/illera88/Ponce>
- "AngryGhidra: Ghidra Plugin based on Angr", <https://github.com/Nalen98/AngryGhidra>
- "SENinja: A symbolic execution plugin for Binary Ninja", L. Borzacchiello et al., 2022

5.3 Voraussetzungen

Sehr gute Kenntnisse IT Sicherheit, Linux, Qemu, GDB, Programmierung incl. Low-Level.