Concolic Firmware Emulation using Dynamic State Selection

Luke Serné

Delft University of Technology

TUDelft

Concolic Firmware Emulation using Dynamic State Selection

by

Luke Serné

at the Delft University of Technology,
to be defended publicly on the 29th of November 2022

Student Number:	5391636	
Thesis Number:	Q&CE-CE-MS-2022-05	
Project Duration:	January 2022 - November 2022	
Thesis Committee:	Dr. Sicco Verwer	TU Delft, Advisor
	Dr. Mottaqiallah Taouil	TU Delft, Supervisor
	Jeffrey Rongen	NFI, Co-supervisor

This research was conducted together with Netherlands Forensic Institute (NFI).

Preface

Before you lies the product of roughly a year of work. I am proud of the work that went into creating this and relieved that it is now complete. During this thesis project, from January 2022 to November 2022, I have learnt a lot, both about technical matters, and about my personal academic preferences.

First, I would like to thank my supervisor Dr. Mottaqiallah Taouil, for the advice and support he gave throughout this project. Additionally, I owe a big thank you to Jeffrey Rongen for his technical knowledge, inspiration, feedback on earlier versions of this document and many fun coffee breaks. Finally, I want to thank Dr. Sicco Verwer, for his advice and guidance.

During the summer of 2021, I followed a workshop organised by the Netherlands Forensic Institute, given by Jeffrey Rongen, on mobile device hacking. This caught my interest, and while talking afterwards, I mentioned I was looking for a company to do my Master Thesis at. He then offered me to do my thesis at the NFI, which led to many months of collaboration on this project. His cheerful presence made every visit to the NFI a joy. The people I met at the NFI have all been very kind and helpful in completing this goal, and I remember these interactions with great appreciation. In particular, I would like to thank Dana, Eljakim, Ginger and Nico for their help and input on my project.

Finally, I would like to thank my friends and family for their relentless support. Especially Linus and Cassie have been very helpful in facilitating the completion of various procedures. Furthermore, I would like to thank Armin, Aslı, Charlie, Marco and Sven for being part of the journey to a master's degree, both as fellow students and as friends. I thoroughly enjoyed our many conversations and games, that helped me throughout the tough trail that eventually culminated in this very document. My family has also been an important part of my life, especially when the various COVID-related lockdowns caused me to move back in. Throughout the past 2.5 years, their support has allowed me to persevere, and their company has kept my motivation high. Thank you, mum, dad and Elisa.

Luke Serné
Breda, November 2022

Summary

In recent years, computers have found their way into nearly every part of life. This led to the creation of many embedded devices, which are usually quite different from the more commonly known computers and each other. The cause of this is the diversity in constraints that are placed on these devices, in terms of size, weight, energy consumption and more. It is important to test and analyse these embedded systems, to avoid malicious actors to gain control of these systems, or for bugs in the code running on these systems to cause problems. In contrast to other computers, embedded systems are often tightly integrated with their environment and are designed to control it. However, there still are major challenges when analysing these systems, just because of their tight integration with the environment. When executing the code out of the environment it was intended for, it often functions differently, if at all.

Current solutions attempt to solve this problem in various ways, but they all fail to create a system that is general and flexible (see Chapter 3). As such, this project set out to find a methodology that solves this problem in an automated way, while ensuring the device's code executes accurately.

This thesis presents a methodology to solve this problem in a general and flexible way, using symbolic execution. By using symbolic execution to handle interaction with the environment, the system can realistically and generally emulate embedded devices. Additionally, symbolic states are made concrete after a while, to continue executing concretely. When the system is executing concretely, user-defined hooks can be executed, which can be used to further instrument the process, allowing for great flexibility. Such extensions could be used to attach a debugger to the system, or by redirection output from a serial connection to a file.

We conduct an empirical study to find that the system emulates accurately, reaching a nearly identical result to the same code running on real hardware, using four diverse samples. Additionally, we compare the system against Jetset, a system that attempts to solve a similar problem using just symbolic execution, and find that, although our system is considerably slower, it remains accurate. The difference in performance is primarily due to their system being more tightly integrated. Still, the added flexibility our system provides makes it more suitable to extend with further analyses, and thus might make it preferable in some situations. Finally, we provide the source code of an implementation of the system to allow researchers to continue developing and improving this system.

Contents

Abbreviations

Abbreviation	Definition
ARM	Acorn RISC Machine
CFG	Control Flow Graph
CPU	Central Processing Unit
DMA	Direct Memory Access
DTB	Device Tree Blob
ELF	Executable and Linkable Format
EOL	End Of Line
GDB	GNU Debugger
GPIO	General Purpose Input / Output
GPU	Graphics Processing Unit
HDMI	High-Definition Multimedia Interface
IO	Input / Output
JTAG	Joint Test Action Group
LTS	Long-Term Support
MMIO	Memory-Mapped Input / Output
MMU	Memory Management Unit
NP	Nondeterministic Polynomial time
OS	Operating System
OpenOCD	Open On-Chip Debugger
PBL	Primary Boot Loader
PR	Pull Request
QEMU	Quick EMUlator
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RNG	Random Number Generator
SD-card	Secure Digital card
SMT	Satisfiability Modulo Theories
SoC	System on Chip
SW	SoftWare
UART	Universal Asynchronous Receiver-Transmitter
URL	Uniform Resource Locator
US	United States of America
USB	Universal Serial Bus
USD	US Dollar
VFG	Value Flow Graph

List of Figures

List of Tables

1 Introduction

This chapter briefly introduces the topic addressed in this thesis, its importance, related work and the main contributions. Section 1.1 presents the motivation and relevance of cybersecurity, discusses why embedded systems are especially vulnerable and highlights the need for a generalised system to analyse embedded systems. Section 1.2 reviews the state-of-the-art in embedded system emulation and shows the challenges these systems are facing. Section 1.3 lists the main research question and the three sub-questions that are derived from the main question. Section 1.4 covers the contributions this thesis makes. Finally, Section 1.5 presents the outline of the remainder of this document.

1.1. Motivation

With the rise of the Internet of Things, there is an increasing number of small systems that are very diverse in their capabilities, collectively called embedded systems. Some systems have various motors, while others have mostly sensors. With the capabilities of these devices increasing, the impact of bugs in the code that powers these devices has also grown. Notable examples of simple bugs that caused large amounts of damage are listed below.

Ariane 501
The Ariane 501 rocket was launched by the European Space Agency on the 4th of June 1996. After about 40 seconds, the rocket broke up and exploded. The cause turned out to be a software bug.

A data type conversion failed in one of its inertial reference systems, and the resulting error code was misinterpreted as the orientation of the rocket. Needless to say, this orientation was very different from the intended orientation of the rocket, so the rocket used booster pulses to 'correct' its orientation. However, this damaged the connection between various systems, which in turn triggered the self-destruct mechanism. [35]

Although the rocket was part of an unmanned mission and as such, no lives were lost, the economical losses were quite large: approximately 370 million USD. [37]

Therac-25
Another example is the Therac-25 radio therapy system that was used to treat patients with cancer. Between 1985 and 1987, seven patients received a severe overdose of radiation, which led to four deaths and three patients who had limbs amputated.

The cause was quite simple. While configuring the device, a lab technician could switch between various modes. As soon the x-ray mode was selected, the machine would begin setting up the system for high-powered x-rays, a process that takes about 8 seconds. If the lab technician changed the mode within those 8 seconds, these were not updated correctly, leading to the system being in an unknown state, which caused the cases of radiation overdose. [13]

Modern Mobile Phones
In 2017, Hay [22] published research that showed many vulnerabilities existed in the bootloaders of

modern mobile phones, such as in the Nexus 9 and OnePlus 3/3T. These bootloaders are the first pieces of code that run whenever a phone is turned on. As such, they are responsible for starting the various parts of the phone, and executing the next stage of the boot process. To make sure that no malicious code is run, the bootloader should also verify that the next stage is trustworthy. However, vulnerabilities in the bootloader can enable an attacker to disable or evade these checks, allowing the attacker to run malicious code while the device is starting up.

To reduce the impact of similar bugs, it is important that these bugs are identified and fixed. Identifying such bugs can be done in several different ways, using different forms of analysis. There are two broad categories of analysis: static analysis and dynamic analysis. The former comprises all analysis techniques that do not execute the code that is under analysis, whereas the latter category comprises those techniques that do execute the code under analysis. This work focuses on the latter category.

Various dynamic analysis techniques have been shown to be quite effective at finding bugs in software for regular computers. [50] Applying these techniques to embedded systems is attractive, and many attempts to do so have been made. [34]

One of these techniques is *Fuzzing*, which feeds random inputs to a program, looking for unintended behaviour. For this technique, the code of a program needs to be executed in a controlled environment. An optimisation can be made by running the program until the first time it reads input, and saving the current state of the program before reading the fuzzing data. Then, when restarting the program for a new input, the execution does not start from the beginning, but instead from the previously saved state before the first input.

Unfortunately, in the specific case of bootloaders, the code that sets up the initial state is surrounded by peripheral communication. As such, problems arise when this communication is attempted in the controlled environment. These problems can be divided into two categories. The first group of problems exists because the emulated environment does not have memory mapped to the address range of the peripheral in question. The second group of problems arises because it is unknown what values should be read when interfacing with this memory. As such, there is a need for a system that intercepts this peripheral communication and handles it gracefully.

1.2. Difficulties in analysing embedded devices

When dynamically analysing the code running on embedded devices, there are various difficulties that are explained below. These difficulties call for a solution that can handle communication with peripherals, without any knowledge about the peripheral in question. In Section 1.2.1, we give a short overview of the existing solutions to this problem. In Section 1.2.2 and Section 1.2.3, we will discuss two reasons why only a single group of solutions is suited to create a general and flexible system.

1.2.1. Current Solutions

Roughly speaking, current solution fall into one of three categories. The first category consists of solutions that pass peripheral interaction on to real peripherals. These solutions require connecting the peripherals to the system the analysis is conducted on. This is a large amount of effort that prevents researchers from quickly testing multiple systems. The second category consists of solutions that emulate or model the peripheral. Emulating the peripheral requires identifying all peripherals present in an embedded system and obtaining their firmware. Modelling requires obtaining a description of the behaviour of all peripherals: how they respond to certain inputs. Both these solutions require a large amount of work when switching between different embedded systems. The third and final category consists of solutions that use symbolic execution to infer what values the program running on the CPU is expecting. While these solutions have the potential to be generally applicable, no such solutions have been published yet. Another important consideration is that the use of symbolic execution largely limits the additional analyses that can be performed.

A more detailed description of these different categories of solutions that have been proposed by previous work, including their limitations, is given in Chapter 3.

1.2.2. Diverse Peripherals

Just the firmware code of the system that is to be analysed is not enough to simulate the system. This is because this code regularly communicates with the various peripherals that are connected to the system. As such, the behaviour of the peripherals, or at least the communication with the main CPU, also needs to somehow be taken into account. When this communication is not taken care of, the system will not do anything useful at all, and will instead get stuck, trying to communicate with peripherals that do not reply (correctly).

However, the diversity in peripherals is a major challenge. Peripherals can vary from a memory management unit (MMU), to crypto engines, to serial connections. These different peripherals clearly behave differently, and even different versions of the same peripheral can behave differently. This wide variety of different peripherals that it is unfeasible to model the behaviour of individual peripherals [14].

1.2.3. Unknown Peripherals

The peripherals that a device has, are registered in Device Tree Blob (DTB) files. Still, there are peripherals that remain unknown. This is because the DTB file format is not intended to present a complete list of what types of peripherals a device has. The peripherals listed in the DTB files are generally only those that the kernel needs to know of, and it only lists the memory ranges of each peripheral, not necessarily its type and model. The existence of these unknown peripherals makes it even harder to model the behaviour of peripherals based on their type and model, and as such, a peripheral-agnostic method is required.

1.3. Research Questions

The aim of this thesis project is thus to explore the possibility of a system that makes it easier to analyse embedded systems.

How can dynamic analysis of embedded system firmware be automated to aid reverse engineering?

To answer the research question, the following sub-questions can be defined:

1. How can peripheral communication be handled automatically?
2. How can peripheral communication be handled accurately?
3. How can good and bad states be distinguished?

1.4. Contributions

In this thesis, the primary objective is to develop a flexible approach to emulate embedded systems in an automated fashion, that can easily be extended by analysts to suit their specific needs. The secondary objective is to create a system that handles the peripheral communication an embedded system performs in a way that is accurate to real hardware.

The contributions of this thesis can be summarised as follows:

- **A novel and flexible approach to emulating embedded systems**: We developed an approach that uses a combination of both concrete and symbolic execution to model peripheral interaction on embedded systems. This combination mitigates the problem of state space explosion and allows further extensions to the emulator to easily prepare the system for fuzzing or other analysis techniques.
- **The successful demonstration of emulating an embedded system using multiple state definitions**: Our approach uses multiple different definitions of what symbolic states are considered suitable for concretisation. This novel approach dynamically changes the definition that is used based on previous performance of that definition. We show that this approach can be used to successfully emulate six different samples, intended for significantly different systems.
- **The successful demonstration of the accuracy of the embedded system**: We show that using the proposed approach, we can not only successfully emulate an embedded system, the peripheral communication that the symbolic execution finds, is accurate and in most cases identical to peripheral communication on the real hardware. That is, the resulting state of the system

is identical between the emulation approach and when executing on real hardware.

- **The implementation of generic system**: Our approach is completely generic and makes no assumptions about the specific hardware of an embedded system. This generality allows an analyst to employ this approach for many different embedded systems without any overhead. This is especially helpful considering the wide variety of embedded systems. We have implemented our approach and made the approach freely available as an open source project. This allows future work to build on top of this work.
- **Several bugs in open source software**: Over the course of this thesis project, several bugs were found in various open source systems, which were reported and patches were submitted. These systems are `qiling` (6 issues) and `claripy` (2 issues), which are the emulation and symbolic execution libraries that this work depends on, respectively. These bugs are listed separately in Appendix A.

1.5. Outline

The structure of this thesis document is as follows: In Chapter 2, we present an overview of knowledge and techniques required to understand the contributions of this thesis and related works. In Chapter 3, we present existing works that relate to the work presented in this thesis. In Chapter 4, we motivate our design decisions. In Chapter 5, we present our system, a peripheral-agnostic emulator. In Chapter 6, we evaluate our system. Finally, in Chapter 7, we conclude this work by discussing the system, evaluating our research questions and providing directions for future research.

2 Background

In this chapter, a detailed description of the terms and concepts that are used in this thesis is provided. These terms and concepts are important for the understanding of the work that is performed and discussed in this thesis. Specifically, in Section 2.1, primary bootloaders are discussed. In Section 2.2 and Section 2.3, the two main different forms of program analysis are described: static and dynamic analysis respectively. In Section 2.4, symbolic execution and its limitations are explained. Section 2.5 describes and compares several methods to compare different instruction traces.

2.1. Primary Boot Loaders

Primary Boot Loaders are the first pieces of code that run when a system is turned on. [25] These loaders are usually very small, especially compared to programs that run in an operating system, and stored on the read-only memory or die of the chip. Their job is to initialise various peripherals and the bare minimum of the memory and storage of the system that is required for further stages of the boot chain. Finally, it loads the next stage in the boot chain.

Since Primary Boot Loaders are the first pieces of code that are run on a system, their environment is quite different compared to other programs. For example, other programs communicate with the underlying operating system to perform various tasks. Additionally, a regular program specifies exactly what regions of memory it will access, and those regions of memory are available exclusively to that program (and the operating system kernel). As such, external influences on these memory regions can mostly be excluded in those cases.

Primary Boot Loaders on the other hand, do not specify beforehand what regions of memory they use. In addition, this memory is often shared with external peripherals, in the sense that both the peripheral and the bootloader have access to these regions and may read and modify the memory within asynchronously. These peripherals are separate systems that are mapped to a specific part of the address space, and operate on the values in this part of the address space asynchronously. These peripherals can have various functions. For example, they can be temperature sensors, random number generators or the memory management unit, which manages external memory.

2.2. Static Analysis

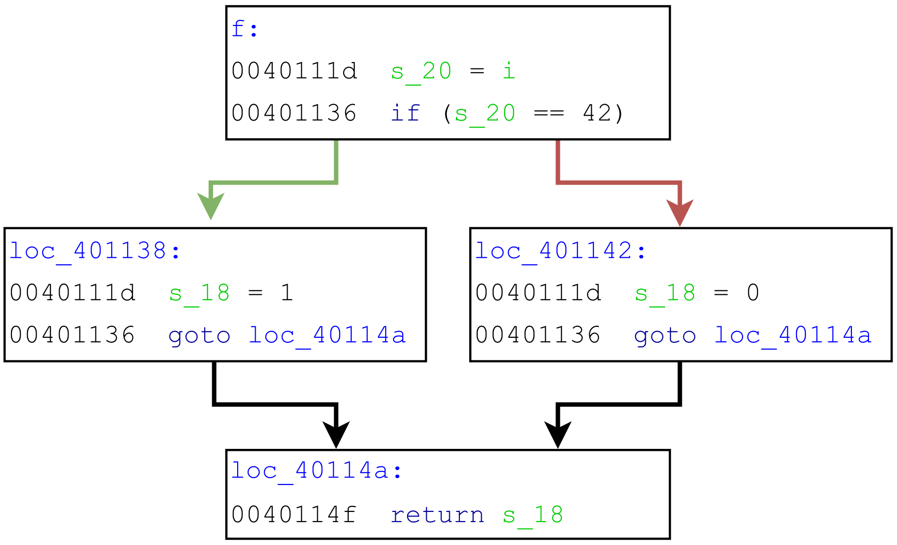
This section discusses various static analysis techniques. This term refers to techniques that attempt to find information about a program without executing it. Often, this involves modelling the code contained in the program binary in some form and reasoning about this model. Two methods that belong to this group of analyses are explained in more detail. Control Flow Graphs are explained in Section 2.2.1 and Value Flow Graphs are explained in Section 2.2.2.

```
def f(i: int) -> bool:
    x = 42

    if i == x:
        b = True
    else:
        b = False

    return b
```

Listing 1: A simple function that returns a value based on an input.



```
f:
0040111d   s_20 = i
00401136   if (s_20 == 42)

loc_401138:
0040111d   s_18 = 1
00401136   goto loc_40114a

loc_401142:
0040111d   s_18 = 0
00401136   goto loc_40114a

loc_40114a:
0040114f   return s_18
```

Figure 2.1: Control Flow Graph that corresponds to the code in Listing 1.

2.2.1. CFG analysis

Control Flow Graph (CFG) Analysis attempts to find out how the execution “flows” through the code and represents this in a graph. More precisely, a control flow graph represents all possible paths the instruction pointer can take in a graph form.

Most instructions only have one possible successor - the instruction placed directly after it. An important exception are branch and call instructions, or instructions that modify the instruction pointer. When these instructions are executed, they change the instruction pointer, causing the next instruction to be executed to possibly not be the instruction placed directly after the current instruction. It is even possible that the actual next instruction depends on the program state.

For example, consider the `if` check in Listing 1. Note that this example does not use an assembly language, but a different language to make the code easier to understand.

As can be seen in the example, the path that is taken through a program is often some part that is executed linearly, followed by a jump to a different part of the code. The control flow graph corresponding to this code is shown in Figure 2.1.

2.2.2. VFG analysis

Value Flow Graph (VFG) Analysis is similar to Control Flow Graph analysis, but instead of tracking how the execution flows through the program, it tracks how data flows through the program [45]. When data is found at a specific point in the program, it might not be immediately obvious how this data got there, and what other values influence a certain value. An example of a VFG for the expression `z = (1 + y) * tan(x)` is shown in Figure 2.2.

Another use for VFG analysis is answering the question of which values a specific variable can take.

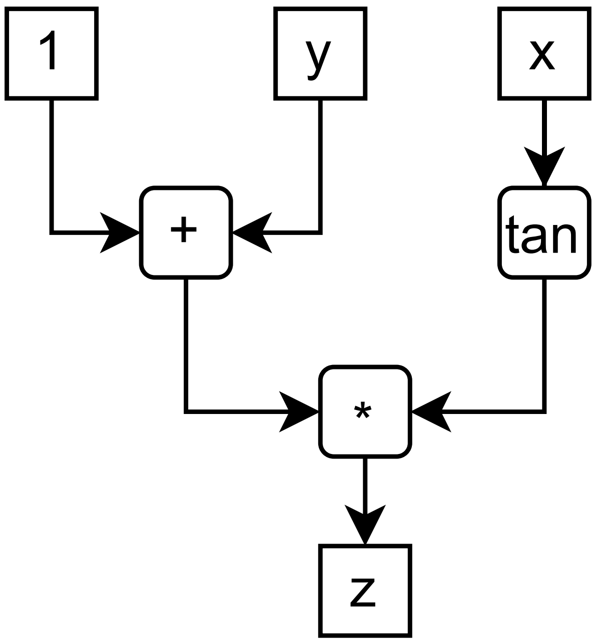


Figure 2.2: Value Flow Graph that corresponds to the statement `z = (1 + y) * tan(x)`.

This question can arise when constructing a CFG of code where the next instruction depends on the value of a register. An example of this is shown in Figure 2.3 and Figure 2.4.

2.3. Dynamic Analysis

Dynamic Analysis is different from Static Analysis in that it also executes (some part of) the code that is under analysis. We discuss two forms in more detail, namely execution in Section 2.3.1 and emulation in Section 2.3.2.

2.3.1. Execution

One form of Dynamic Analysis consists of running the code on the system it is intended for, and observing the effects of the code. Usually, computers execute code very quickly. That is, they execute many instructions within a single second. When trying to identify the effects of specific parts of the code, it is helpful to be able to stop the execution at arbitrary points in the program and investigate the state of the program at these points. To this end, many different tools have been developed. These tools, commonly known as ‘debuggers’, allow an analyst to, among other things, stop the execution of code at specific instructions, to inspect (and sometimes modify) the values in the system’s memory and registers.

GDB

One such debugger is GDB, short for GNU Debugger. This debugger can debug programs that run on the same system, but it also supports to debug programs remotely. In this mode, it sends the debugging commands over a network or serial connection to the device that the target program runs on. Although this debugger has a wide array of features, the work in this thesis primarily uses three.

Breakpoints

Breakpoints are points at which the code execution stops. In other words, when the processor tries to execute the instruction at the location of a breakpoint is, it instead stops executing. This is useful for quickly running the code of a program where the precise effects of individual instructions are deemed uninteresting, but where the collective result of all those instructions is more interesting. In this case, a breakpoint can be placed just after the last instruction of the group of instructions that produces an interesting result.

```python
def f_0() -> int:
    return 2
def f_1() -> int:
    return 1
def f_2() -> int:
    return 0

def f(x: int) -> int:
    x = minimum(x, 2)

    g = list(f_0, f_1, f_2)[x]

    return g()
```

Figure 2.3: A simple function that returns a number based on an input number.

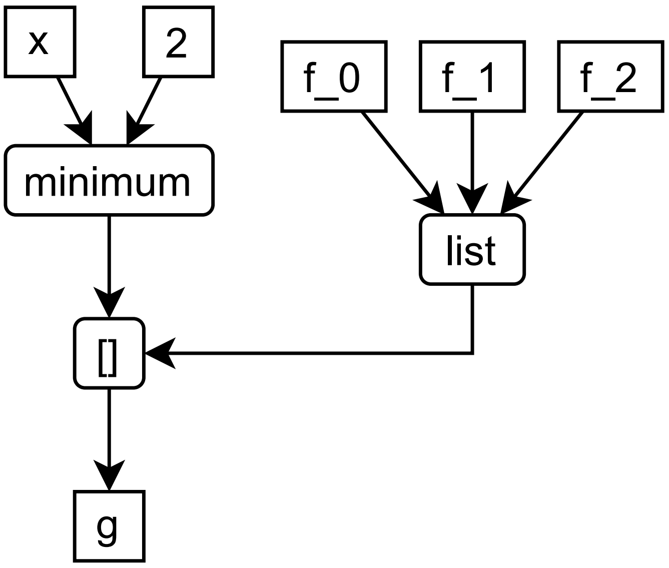


Figure 2.4: Value Flow Graph that corresponds to the variable `g` in the function `f` in Figure 2.3.

Stepping
Stepping through code refers to executing instructions one-by-one, and waiting for input from the analyst before executing the next instruction. This allows the analyst to slowly execute a part of the code that the analyst is interested in. As such, the effects of individual instructions can be analysed.

Register reading
Reading the values of the registers of the processor is an important feature. This gives insight into the internal state of the processor. For example, an analyst might expect a register to hold a specific value before executing a specific instruction. Using breakpoints, stepping and reading the value of the register just before the specific instruction, the analyst can verify this expectation.

JTAG
JTAG, named after the Joint Test Action Group that standardised it, is a standard that describes a protocol that chips can follow to allow for debugging. The precise workings of the protocol are beyond the scope of this background section. However, it is relevant to know that the inclusion of a JTAG interface has become widespread, especially among chips on boards that are geared towards development. For example, models of the Raspberry Pi can provide a JTAG interface on some of their GPIO pins if the correct configuration variables are set.

When the interface is enabled, a device such as the J-Link can be connected to the correct pins, and serve as the bridge between a computer that an analyst uses and the board under investigation. Then, software such as OpenOCD or J-Link Commander can be used on the computer to communicate with the J-Link device. Finally, the software opens a connection for a debugger such as GDB, so the analyst can connect GDB to the board. This then allows the analyst to perform actions such as breakpoints, stepping and register reading, as described above.

2.3.2. Emulation

Emulation refers to running code on a different system than what it is intended for [2]. At the lowest level, all code is written in an assembly language that can only be understood by chips of a specific type (architecture). When analysing programs that are designed for chips of a different architecture, Dynamic Analysis appears impossible. After all, the CPU simply does not understand the assembly language the program uses and as such, it cannot directly execute the code.

This problem can be solved in several ways. One way is to simply recompile the program's source code for the CPU that one wants to perform the (Dynamic) Analysis on. However, this requires access to the source code, and, more importantly, there is no guarantee that the behaviour of the program on the original CPU is the same as the behaviour on the CPU of the machine that is used for analysis.

A solution that avoids both these problems is to translate the individual instructions to (possibly multiple) instructions the CPU used for the analysis can understand. This approach has several drawbacks. First of all, this translation takes time. Second of all, the original code was likely optimised by the compiler. However, there might be operations that cannot be performed using a single instruction in the original instruction set. So, the compiler had to use several instructions that, when executed after one another, perform the desired operation. When emulating this on a per-instruction level, each of these instructions would result in (possibly) several instructions, even though there might be a single instruction that achieves the originally intended operation. These extra instructions do not contribute anything, yet they do cost time to execute. Even when the emulator attempts to account for this effect, the problem cannot be fully eliminated. As such, emulation is often several times slower than the native code [2].

2.4. Symbolic Execution

This section explains symbolic execution and its limitations. The general methodology of symbolic execution is explained in Section 2.4.1 and its limitations are discussed in Section 2.4.2. Finally, Section 2.4.3 describes several solutions that have been proposed to the limitations of symbolic execution.

2.4.1. Method

Symbolic Execution is a method of executing a program, where variables whose values are not known at execution time are treated as symbolic vectors. Although the concrete values of these vectors are not known, some properties of the value the vector represents are known. For example that it is a 32-bit integer value. These restrictions on the possible values of a variable are called constraints. Apart from the initial definition of the variable, later uses of it might put further, more strict constraints on the value of the variable. If several constraints are combined, it might happen that only a single value is possible.

However, there regularly are different paths that could be taken. Which path is taken depends on the value of the variable. When this happens, the current state is duplicated and all possible different paths are explored by one of the duplicated states. Each of these states gets an extra constraint that ensures that the variable satisfies all conditions to follow that specific path.

To illustrate this, consider the example in Listing 2. Suppose it is unknown what the value of `i` is, but it is known that `f(i)` should return `True`. Using symbolic execution, a corresponding value of `i` can be found as follows:

1. Introduce a new variable i for state s_0 that represents the function parameter `i`.
2. Create two new states that inherit from s_0: s_1 and s_2 that correspond to the two branches created by the `if` on line 2. Let s_1 correspond to the 'true'-branch, and let s_2 correspond to the 'false'-branch. That means s_1 gets an additional constraint of $i = 13$. Similarly, s_2 gets an extra constraint of $i \neq 13$.
3. Continuing the path of s_1, a `return` statement occurs, with a return value of `True`. This means this state reaches the desired return value. Since the goal is to find a fitting value for the input i, the constraint for i on the state s_1 can be given to a constraint (SMT) solver. In this case, the constraint is quite simple ($i = 13$), so it is obvious that the only possible input to `f` that returns `True` as a result is the value 13. Thus, this must be the value of the variable `i`.

```
1 def f(i: int) -> bool:
2     if i == 13:
3         return True
4     else:
5         return False
```

Listing 2: A simple function that returns a value based on an input.

2.4.2. Limitations

Although symbolic execution is very powerful, there are several weak points that have been investigated in the literature. These weak points are:

1. **State explosion**. As shown in the previous example, the number of states doubles at every `if` that depends on a symbolic variable. When loops are encountered, whose exit condition depends on a symbolic variable, the number of states can grow even further. To exemplify this, consider the example code in Listing 3.

```python
def f(x: list[int]) -> bool:
    sum = 0
    for value in x:
        sum += value

    if len(x) == 100 and sum == 42:
        return True
    else:
        return False
```

Listing 3: A simple function that returns `True` if and only if the input list of numbers sums to 42 and has 100 elements.

   There are infinitely many paths through this code - after all, the number of iterations of the loop depends on the length of the input parameter. Searching through these infinitely many paths to find a path that leads to `True` being returned will take a very long time.

2. **Complex constraints**. The constraints that have been shown in the previous examples were quite simple, because the programs were quite simple as well. However, most programs are considerably more complex and give rise to complex systems of constraints.

   Solving a system of constraints is equivalent to the satisfiability problem, which has been shown to be NP-complete [8]. In fact, while satisfiability only considers boolean variables, the constraints symbolic execution generates have numbers or vectors of numbers as variables. This additional complexity makes the problem even harder to solve.

   However, since this problem is quite well-known, and in fact appears in many fields of study, several programs have been developed that can solve (some) systems of equations within polynomial time [1, 12]. Especially for simple systems with few equations and variables, these programs can give a solution within a reasonable amount of time.

3. **Execution Performance**. Symbolic execution is typically much slower than concrete execution, because all operations need to be modelled. Often, a single instruction that can be executed very quickly on the original chip results in many steps for the symbolic execution engine. Applying those steps further complicates the existing constraints, and making subsequent instructions even more time-consuming to execute.

2.4.3. Proposed Solutions

This subsection explains several different solutions and mitigations that have been proposed by various authors to the limitations of symbolic execution.

State explosion

One of the solutions that have been proposed for the problem of state explosion was proposed by Kuznetsov et al. [32]. Their solution is to merge states when their paths join together. Even though there are still equally many paths, the number of states can be drastically reduced.

For example, consider the function in Figure 2.5. A CFG corresponding to this function (Figure 2.6) shows that there are four paths through this function. However, there also are two merge points that all paths through the function pass through. When the constraints of the incoming paths are combined at such a merge point, the symbolic execution can continue from that point using only a single state. Doing that for this function results in a single state that reaches the return statement instead of four states.

The downside to this improvement is that the constraint of this single state is more complex than each of the four states' constraints. This means the constraint solver will take longer to solve this constraint.

Complex constraints
The problem of complex constraints can be mitigated by summarising specific parts of a program that are particularly complex. This summary means that the complexity of that part of the program only needs to be evaluated once, after which the simplified form can be re-used.

For example, Godefroid [17] use this technique to create a symbolic expression for the outputs of a function in terms of its inputs. This then allows their algorithm to more efficiently execute the function(s) that use this summarised function.

Still, a particularly complex function (such as a hash function) might be too complex to summarise effectively. Another possibility is that the symbolic executor is not powerful enough to process specific constraints. This might lead to imprecise results, where the constraints that are generated as a summary do not model the underlying code perfectly.

Execution Performance
A solution to the problem of poor execution performance because of the overhead associated with executing instructions symbolically is proposed by Gritti et al. [18]. Their solution combines symbolic and concrete execution by executing concretely as much as possible and only switch to symbolic execution when needed.

In their approach, they let users define an initial starting point, a point to switch to symbolic execution, and a final point of interest. Their system will run the program concretely from the starting point until the point where the user specified to switch to symbolic execution. Here, the user can choose to make various things symbolic, for instance the return value of a function. Then, the concrete state is carried over to the symbolic execution and the remainder of the execution happens symbolically. This continues until the point of interest is reached. At this final point, the user can investigate the results of the symbolic execution.

Although this effectively eliminates the problems associated with symbolic execution for the part before the point at which the symbolic execution starts, this approach still suffers from the same problem while performing the symbolic execution. It should be noted that the starting point of the symbolic execution might be quite far removed from the point of interest, which will inevitably result in many different states.

2.5. Comparing Instruction Traces

An instruction trace is a sequence of numbers, specifically the addresses of the instructions that are executed, in the order in which they are executed. Comparing the instruction traces between different emulators and real hardware is helpful in evaluating how well the execution within different emulators corresponds to the execution on real hardware. Ideally, every piece of code should execute identically in an emulator and on real hardware. To measure this, several metrics can be used.

2.5.1. Jaccard Index

The Jaccard Index is a measure of how alike two sets of items are. It is a value between 0 and 1, which is computed for sets A and B using the formula $\frac{|A \cap B|}{|A \cup B|}$ [16, 26, 47]. In words: the number of items that appear in both A and B, divided by the number of items that appear in either A or B. The highest possible Jaccard Index is 1, for two sets that contain the exact same elements. The smallest possible Jaccard Index is 0, for two sets that contain no elements in common.

For example, if the set A is $\{a, b, c, d\}$, and the set B is $\{c, d, e, f\}$, there are two common elements (c and d), and the total number of unique elements is 6 (a, b, c, d, e and f). As such, the Jaccard Index of A and B is $\frac{2}{6} = \frac{1}{3}$. This example is represented visually in Figure 2.7.

Note that this metric works on sets, which means that the order and multiplicity of the items are ignored. As such, it is not well-suited for comparing sequences of items. To illustrate this, consider we want to find the sequence that is the closest to abc from $\{aabc, cba, aabbc\}$. Intuitively, $aabc$ would be closest to abc. When computing the Jaccard Index between abc and each of the candidates, we get 1 for each of them. This implies that they all are the same as abc, but this is clearly not the case.

```python
def f(x: int) -> int:
    if x == 2:
        y = 8
    else:
        y = 5

    if x + y < 10:
        y = 11
    else:
        y = 8

    return y
```

Figure 2.5: A simple function that returns a number based on an input number.

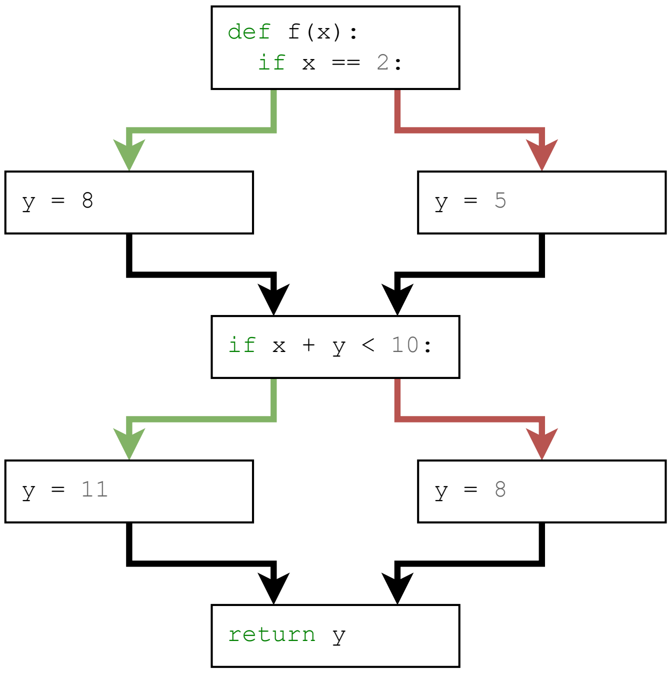


Figure 2.6: Control Flow Graph that corresponds to the code in Figure 2.5.

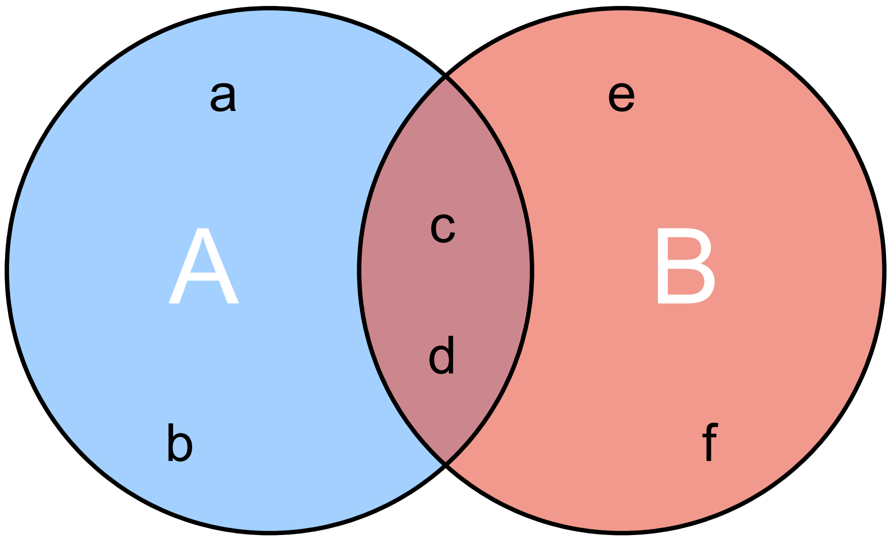


Figure 2.7: A graphical example of the Jaccard Index of the sets $A = \{a, b, c, d\}$ and $B = \{c, d, e, f\}$. The Jaccard index is $\frac{|\{c,d\}|}{|\{a,b,c,d,e,f\}|} = \frac{2}{6} = \frac{1}{3}$.

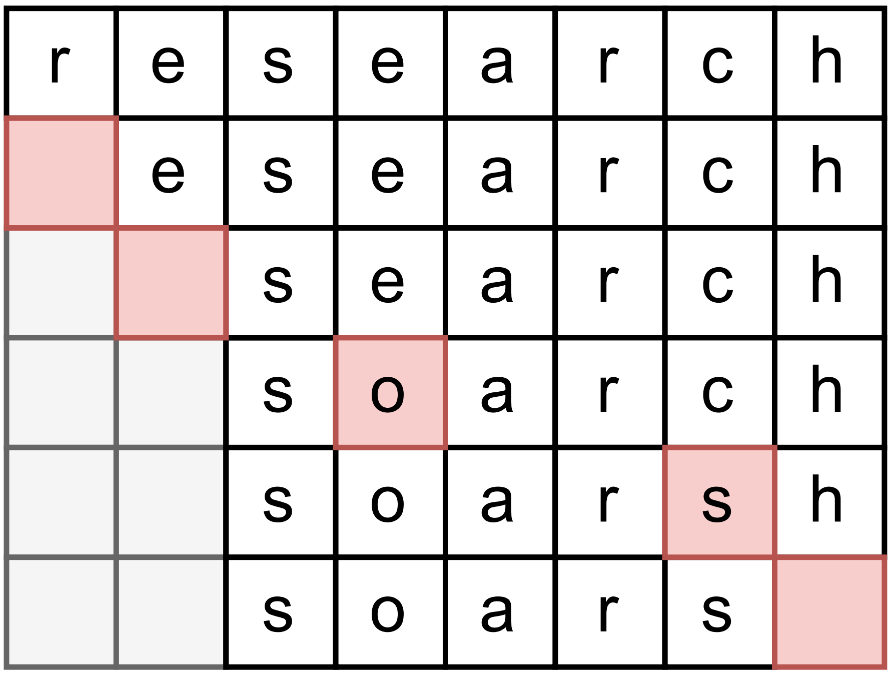


Figure 2.8: A graphical example of the Levenshtein Distance between the sequences $research$ and $soars$.

2.5.2. Levenshtein Distance

When comparing two sequences of items, their similarity can be expressed by their Levenshtein Distance [51]. Put simply, this measures the number of single-element insertions, deletions or changes that are required to change one sequence into the other. As such, the smallest value is 0, indicating the two sequences are identical, and there is no upper bound. The larger this distance is, the less similar the sequences are.

For example, the sequences $research$ and $soars$ have a distance of 5, because 5 edits can be used to change $research$ into $soars$, and it cannot be done with fewer edits. First, two edits are required to remove the re at the beginning of $research$. Next, the remaining e has to be changed into an o, bringing the number of edits to 3. Additionally, the c has to be changed to s, and the final h has to be removed. Doing this brings the total number of edits to 5. The full example is represented visually in Figure 2.8.

The Levenshtein metric is much better suited to quantify the difference between two sequences than the Jaccard Index. Consider the same example as before, where we wish to find the sequence that is the closest to abc in $\{aabc, cba, aabbc\}$. Now, the Levenshtein Distances are 1, 3 and 2 respectively. This clearly indicates $aabc$ is the most similar to abc, which agrees with our intuition.

3 Related Work

Over the years, many systems have been developed to analyse the code running on embedded systems. In general, these systems can be seen as using a hardware-in-the-loop approach, a pure emulation approach, a symbolic execution approach or a combination of these. Section 3.1 provides an overview of the different systems and how they fall in these different categories and compares the different categories to find an approach that can answer the research questions. Section 3.2 provides an explanation of the methodology of systems that use a pure emulation approach. Similarly, Section 3.3 provides an overview of the methodology of systems that use a hardware-in-the-loop approach. Finally, Section 3.4 explores the systems that use symbolic execution in more detail, also providing a summary of each of the published works.

3.1. Overview

The existing work that is most closely related to this research has been published in a field called “rehosting”. The research in this field aims to “rehost” executable files designed for a certain system on another system. More concretely, the research often focuses on running user-space programs designed for embedded devices such as routers on general-purpose computers such as laptops.

The main challenge these papers have to solve deals with the different environments of these systems. Embedded devices often have a multitude of attached sensors, actuators, internal clocks, cryptographic modules and other systems that are directly related to the functioning of the system (collectively called ‘peripherals’) that the user-space programs running on them directly communicate with. General-purpose computers either do not have these peripherals, or there is an operating system level abstraction that means they cannot be communicated with directly. This layer of abstraction leads to standardised interaction with peripherals, which can be used by analysis tools. As such, this challenge of peripheral communication is exclusive to embedded systems, where no such abstractions are present.

The goal of these papers is to emulate the aforementioned programs on a general-purpose computer, often with the additional goal to discover vulnerabilities in the emulated system through fuzzing. Some papers, such as [40], only have the goal to emulate embedded systems, whereas other papers, such as Shoshitaishvili et al. [44], only emulate a small part of the system. In the case of Cao et al. [4], only the part that deals with authentication is of interest.

In 2021, Fasano et al. [14] released a paper providing an in-depth overview on the state of the art in this field. Table 3.1 is adapted from this paper and extended with several works that have been released since Fasano et al. published their paper. This table shows how the different works in this field compare. The table makes a distinction on how the different components of an embedded system are handled by different papers. The table also categorises the papers in four broad categories, of which three are distinct. These three distinct categories are named Pure Emulation, Hardware in the Loop and Symbolic Abstractions and are discussed in Section 3.2, Section 3.3 and Section 3.4 respectively.

	System	Layer				Target	
		Hardware	OS	Application	Function	ISA	Binary
Pure Emulation	BaseSafe [38]	●	●	●	⊠	ARM	✓
	Clements et al.[7]	⊠ ●	●	○	○	ARM	✓
	Costin et al. [10]	⊠	●	○	○	ARM, MIPS	✓
	DICE [39]	⊠ ●	○	○	○	ARM, MIPS	✓
	Firm-AFL [50]	⊠	●	○	○	ARM, MIPS	✓
	Firmadyne [5]	⊠	●	○ ●	○	ARM, MIPS	✓
	FirmAE [30]	⊠	●	○ ●	○ ●	ARM, MIPS	✓
	FirmWire [24]	⊠	○	○	○	ARM, ARM-Cortex A	✓
	HALucinator [6]	⊠ ●	●	○	○ ●	ARM	✓
	Li et al. [34]	●	●	○	○	—	×
	LuaQEMU [42]	⊠	—	○	○ ●	ARM	✓
	P2IM [15]	⊠ ●	○	○	○	ARM	✓
	PartEmu [21]	⊠ ●	○	○ ●	○	ARM	✓
Hardware in the loop	Avatar2 [40]	↔ ⊠ ●	—	↔ ○	↔ ○	ARM	✓
	Charm [46]	↔ ⊠	●	○	○	ARM	×
	FEMU [33]	↔ ⊠	○	○	○	x86	×
	FirmCorn [19]	⊠	↔ ●	○	○ ●	ARM, MIPS, x86	✓
	Frankenstein [43]	⊠ ●	●	○	○ ●	ARM	✓
	Kammerstetter et al. [28]	↔ ⊠	↔ ●	○	○	MIPS	✓
	Pretender [20]	↔ ⊠ ●	—	○	○	ARM	✓
	Prospect [29]	↔ ⊠	↔ ●	○	○	MIPS	✓
	Surrogates [31]	↔ ⊠	○	○	○	ARM	✓
Hybrid Approaches	Avatar [49]	↔ ⊠ x	↔ ○	↔ ○	↔ ○ x	ARM	✓
	Inception [9]	↔ ⊠ x	○	○	○	ARM	✓
	Mousse [36]	↔	↔ ○	○	○ x	ARM	✓
Symbolic Abstractions	FIE [11]	x	—	○	○	MSP430	×
	FirmUSB [23]	x	—	○	○	8051/52	✓
	Jetset [27]	x	—	○	○	ARM	✓
	Firmalice [44]	x	○ x	○	○ x	ARM, PPC	✓
	Laelaps [4]	x	○	○	○	ARM	✓
	This work	x	—	○	○	ARM(64), PPC, x86, MIPS	✓

↔: Pass-Through
⊠: Emulated
○: Not Modified
●: Replaced
x : Symbolic Model

Table 3.1: Overview of the research present in the field of rehosting, also showing this work for comparison. Adapted and extended from Table 1 in Fasano et al. [14].

Method	Scalable	Realistic values
Hardware in the loop	×	✓
Peripheral Emulation	×	✓
Symbolic Abstractions	✓	?

Scalable: The same system works for different devices, without significant overhead when switching between devices.
Realistic values: The values in the peripheral communication can also be achieved on real hardware.

Table 3.2: Overview of the various approaches to modelling peripheral communication and their performance on two categories.

Paper	Scalable	Concolic Execution	Binary	Available	Assumes OS
FIE [11]	×	×	×	✓	×
Jetset [27]	×	×	✓	✓	×
FirmUSB [23]	×	×	✓	×	×
Firmalice [44]	×	×	✓	✓	✓
Laelaps [4]	×	✓	✓	✓	✓
This work	✓	✓	✓	✓	×

Scalable: The same system works for devices of more than two architectures.
Concolic Execution: The system combines concrete and symbolic execution.
Available: An implementation of the system is freely available.
Assumes OS: The system assumes the code under test contains an operating system (OS) and operates at the operating system level.

Table 3.3: Overview of the research present in the field of rehosting, specifically those that use Symbolic Execution. This table further specifies their limitations.

The overarching problem that the research in this field attempts to solve is the problem of communication with peripherals. Although it is no problem to ignore the data that the CPU sends to the peripherals, it is much harder to answer requests for data from the CPU. After all, answering with data the CPU does not expect will often lead to the CPU entering an error state. Several different methods to deal with this have emerged.

3.2. Pure Emulation

This approach deals with the problem by emulating the peripherals as well. Although this leads to very realistic communication, it increases the amount of work that needs to be done to emulate a system. This method brings with it several new challenges, such as retrieving the code that runs on each of the various peripherals, or even determining the design of the peripheral's hardware.

To alleviate this problem, some research merely models the behaviour of the peripherals based on the communication. This makes adding support for new types of peripherals as simple as adding another configuration file that specifies the responses of the peripheral to various inputs.

However, one major limitation of this method is that it scales poorly to different systems, that often have different peripherals as well. This means that for every new system, several new peripherals need to be emulated or modelled. Although initially, the hope was that, as time passed, one would come across more and more peripherals that have already been implemented, it has been shown that this is not the case. [14]

3.3. Hardware in the loop

This approach deals with the problem by connecting the emulator to real hardware and pass the communication through to the real peripherals. This method clearly results in very realistic behaviour, but its main downside is that it is not always possible or practical to connect the emulator to real hardware. Even if it is possible, a significant amount of effort is required.

This approach is also not scalable, as emulating a new system requires the researcher to connect the necessary hard- and software again, with new peripherals.

3.4. Symbolic Abstractions

This approach deals with the problem by analysing the code that is executed just after the peripheral read, and making an educated guess about the value the CPU expects to read. This guess is based on the structure of the code surrounding the peripheral read.

In contrast to the other two methods, this method has the potential to be more scalable to different types of systems. However, there are several more challenges that come with this method.

- The first is the analysis itself. As the symbolic execution gives the various possible future states of the system, in this method, these future states need to somehow be judged. After all, the effectivity of this system relies on its ability to choose values that avoid error states. Discerning error states from regular states is one of the major challenges in this research.
- The second is the handling of interrupts. Because interrupts can happen after every instruction, in theory, the symbolic execution should fork after every instruction. This further accelerates the state space explosion, severely limiting how far into the future the symbolic execution can actually go.
- The third is choosing a possible value. Even after selecting a single state that the emulation should reach, there often are several different values that can be read that all reach the same state. Although the differences between these different values are not apparent in this state, different values can lead to different paths in the future.
- The fourth is false positives introduced by over-approximation. It is assumed that every peripheral read can result in the full range of values, even though some values might not be possible in practice. This can lead to the analysis of paths that are not realistic, and thus create misleading results. [11]

Other problems this method faces apply more generally to Symbolic Execution, and as such are discussed in Section 2.4.

3.4.1. Works using Symbolic Execution

The differences between these approaches is shown in Table 3.2. Since one of the aims of the research in this paper is to create a system that is generic and scales well to new systems, this paper will focus on applying symbolic execution to the field of rehosting. Because of this, four papers will be discussed in a bit more detail, as they also apply symbolic execution. A comparison of these four papers is given in Table 3.3.

Davidson et al. [11] (FIE) use purely symbolic execution. In addition, their system requires the source code of the firmware that is to be analysed. In fact, Davidson et al. developed a (wrapper around a) compiler that adds instrumentation to simplify the symbolic execution. This instrumentation guides the symbolic execution. The symbolic executor that is used is KLEE [3]. This system is specific to the MSP430 SoC. Because of their use of purely symbolic execution, the FIE system works best on very short samples, and struggles to analyse bigger samples.

Hernandez et al. [23] (FirmUSB) use symbolic execution to discover the program paths that interact with a specific USB device. From these paths, they construct a model of the device. In addition, they use the information about the peripherals to create a model of the expected USB behaviour. Then, these two models are compared, and unexpected behaviours are identified and reported. Unfortunately, they patented their system and did not release their code. To mitigate the problem of state explosion during the symbolic execution, they employ several shortcuts that reduce the generality of their system. As a result, their system only supports the 8051 and 8052 architectures, and is specific to the USB protocol.

Shoshitaishvili et al. [44] (Firmalice) designed a system that uses only static analysis techniques to find authentication bypass vulnerabilities. Their system works on binary files for ARM and PowerPC processors. They statically determine the code between the point where the user gives input, and the authenticated code. Next, they symbolically execute this, to find the constraints for reaching the authenticated code. Finally, they analyse the constraints to see if there is anything odd about them (for instance, hardcoded usernames and passwords). Unfortunately, they did not release their code.

Cao et al. [4] (Laelaps) employ symbolic execution every time a peripheral is read. They mitigate the problem of state space explosion by limiting the number of steps of symbolic execution to a fixed number. By default, this number is 3. They attempt to steer the emulation based on three heuristics. The first is to avoid loops within the symbolic execution. The second is to prioritise new paths. If after applying these heuristics, there still are two final states that appear equally good, the state with the highest address is chosen. The system they developed only supports ARM CPU architectures.

Johnson et al. [27] (Jetset) take a multistaged approach to solving the problem of finding peripheral values to get to a specified target point in a program. Their first step is to symbolically execute the program until the specified point is reached. To enable analysts to continue execution the program in an emulator, they keep track of the values that are used during symbolic execution and use those values in the second step to generate a peripheral definition that can be used in QEMU, an emulator. The third and final step is to run the program from the entry point to the target point. Similar to the work in this thesis and unlike much other work, Johnson et al. [27] aims to provide analysts with a state which can be further executed in an emulator. Because of this similarity, Jetset was chosen as the state of the art for the purpose of evaluating the system proposed in this thesis.

4 Design

This chapter describes the design decisions that were made in designing the approach outlined in this thesis. To answer the research questions, we have to define several terms. In the way we define these, we inevitably make decisions on the final design of the system. Section 4.1 narrows the peripheral communication to read operations by the CPU. Section 4.2 discusses how peripheral reads can be intercepted. Section 4.3 discusses how bad states can be detected when the emulator enters them and provides two implementations. Section 4.4 describes how bad states can be detected during the symbolic execution, and finally, Section 4.5 defines several identifiers of good states during symbolic execution.

4.1. Peripheral Communication

Although the research questions mention peripheral communication, in which information flows both ways, only incoming information is important from the CPU's perspective. This information consists of the data that the CPU reads from the peripherals. These instances are referred to as *peripheral reads* in the remainder of this thesis.

The information the CPU writes to the peripheral does not need to be used in the system, because this information will not influence the way the CPU operates in the future.

4.2. Intercepting Peripheral Reads

The previous design decision reduces the problem of intercepting peripheral communication to intercepting peripheral reads. However, we still have to differentiate between reads from regular memory and reads from peripherals.

To this end, a division should be made between the memory regions where peripherals reside and the regions whose memory belongs to the main CPU. Any read in the former region will be resolved using symbolic execution. Any read in the latter region will be resolved normally. The values written there are stored concretely and those values will be read. These values are part of the *global state*.

This strong division between purely symbolic memory regions and purely concrete memory regions has a limitation. Sometimes, values are read from peripherals and directly stored in the global state. Only much later, the value is read back from the global state, and it is checked for validity. Because the symbolic execution is only used for the first read, an unconstrained value is resolved and it is written to the global state. When this value is later read, the checks on it might fail, which might lead to an error state. Although the backtracking will eventually find the correct value, this will take a very long time.

This problem could be alleviated by having the read resolver component check for this case and then store the symbolic value (with its constraints) instead of a concrete value in a symbolic shadow memory. Then, whenever a read from global state occurs, it first checks if that address exists in this shadow memory. If that is not the case, the value is served from the regular memory. If the address does exist in the symbolic shadow memory, this read is handled like a regular peripheral read, with the added constraints of the symbolic vector that is read from the shadow memory.

4.3. Bad States

The symbolic execution attempts to avoid bad states. We define them as states that contain infinite loops. However, a human analyst should be able to define additional bad states, such as the error handler function. Although bad states theoretically consist of the combination of the entire memory and all registers, bad states can usually also be identified based on the value of the instruction pointer. Although this might detect bad states later than truly optimal, this simplifies the definition process of these bad states considerably.

However, when the system is executing concretely, we might still end up in an infinite loop. This can happen because the symbolic execution did not look far enough into the future. To identify and intercept some of these infinite loops, we use a simple blacklist of opcodes of instructions that encode an infinite loop. These instructions are branch instructions use a hard-coded opcode, based on the instruction set architecture of the code, and make sure the system never executes this instruction. This hard-coded opcode always encodes the instruction that performs a relative branch to itself.

However, more complicated loops cannot be detected using this approach. For that reason, a different system is in place. This system is described in the following subsection.

4.3.1. Loop detection

This component attempts to detect when the emulator has ended up in an infinite loop. To this end, for each instruction that is executed by the emulator, we keep a counter. Every time an instruction is executed, the corresponding counter is incremented. If this number exceeds a global parameter N, the system considers itself stuck, and the Backtracker is activated.

However, the simplicity of this approach comes with the downside that it is counting instructions, not iterations of loops. Some instructions are executed many times, even though they are not loops. This happens in functions that are called several times, with tight loops that iterate many times - for example in `memcpy` or `memset`.

4.3.2. Choosing parameters

The parameter N is configurable. The default values were chosen based on some limited testing, but users can configure these values to their needs. The underlying data structure uses memory linear in the number of instructions that are executed.

A threshold value that is too low might result in falsely identifying branches as being part of loops, and thus cause the system to incorrectly backtrack on correct values. A threshold value that is too high will make the system slower to detect that it is stuck in loops. As such, it is important to pick a good value for N.

4.3.3. Alternative Implementation

Initially, a different system was designed for this component, which was not used in favour of the implementation specified above. The system would have kept track of all branch instructions in a sliding window based on the branch origin and target address. Then, whenever a sequence of elements appeared more than a specific threshold number of times, the system would conclude the emulator was stuck in an infinite loop.

The main difficulty with this alternative implementation lies in determining the size of the sliding window and the threshold number. In addition, the alternative implementation was slightly more complex and as such resulted in more overhead for every executed instruction.

4.4. Finding Error States

The Read Resolver component attempts to avoid error-states. However, while it is sometimes intuitively clear what states are error states, it is non-trivial to distinguish between these states automatically. One definition of an error state could be “any state that can only lead to an infinite loop”. However, this definition fails to consider the possibility that something useful is done before the infinite loop is entered. As such, this definition is too broad. In this work, we use a more narrow definition, to avoid falsely marking states that can still do something useful as error states. We limit the error states to those states that are part of an infinite loop that does nothing but loop.

This definition is quite narrow, and this can lead to undesirable situations. For example, many binaries contain panic handler functions, that are called when the binary has entered a bad state from which it cannot recover. Because this function is not just an infinite loop, but also logs a message and performs some clean up, the function is not marked as indicating an error state. This means the symbolic execution has to process the entire function (and all functions that the error handler calls) only to find it ends up in an infinite loop. Furthermore, if the depth of the symbolic execution is not high enough, the unavoidable infinite loop at the end might not even be found, and the system might be led into the error handler.

Although backtracking will always eventually correct the mistake, it saves a considerable amount of time when the panic handler is avoided altogether. To allow for this, a human analyst can use the Analysis Tool to mark specific addresses, such as the entry point of the panic handler, as error states.

4.5. Good States

Formulating a single definition of a good state that fits all cases is non-trivial. Because of this, we employ several different definitions, each tailored to a specific scenario. When a peripheral read occurs, one of these definitions is chosen. These definitions are called Tactics. The different tactics try to balance avoiding bad states and state explosion in the symbolic execution. Each tactic specifies a success condition of the symbolic execution. This success condition corresponds to a possible definition of a good state. The symbolic executor then tries to find a state that satisfies this condition.

Dummy Tactic

This tactic is the simplest tactic. It just returns a value, defining the goal of the symbolic execution as the current address and accepting any state that reaches this address. This tactic essentially completely bypasses the symbolic execution, which results in a quick answer.

The main advantage of this tactic is that the requirements for the symbolic execution are extremely simple, which enables the symbolic executor to quickly find a possible solution.

This is also the main disadvantage of the tactic. If control flow depends on the read value, then it is likely that a suboptimal path is taken. However, analysis has shown that a large portion of peripheral reads do not influence control flow at all, in which case this disadvantage does not apply.

Return Tactic

The Return Tactic is configured with a specific value `x`. This tactic will attempt to make the function the emulator is currently executing return the value `x`.

To this end, it uses the analysis capabilities of the Analysis Tool to find the instructions that belong to this function and builds a control flow graph of this function. Then, the sink nodes of this graph are the returning basic blocks. These basic blocks constitute the goal addresses of the symbolic execution. Note that special care must be taken to handle functions with a common return footer properly. In case it is detected that a function uses a common return footer, the procedure detecting the return instruction is run on this function as well, recursively.

For every state that reaches one of the detected return instructions, a constraint is added that the returned value equals the value `x`. The returned value is defined as the value of a specific location (stored in memory or in a register). This is not strongly defined, because different calling conventions store the returned value in various places.

A major advantage of this tactic is that the goal addresses are usually close to the address at which the peripheral read happens. This means the symbolic execution is limited, which helps mitigate the effects of state explosion.

Disadvantages of this tactic are that it requires a parameter `x` that is not trivial to determine, and that knowing the calling convention of the current function is required. The latter disadvantage can be mitigated by using heuristics that are built into `angr`. The former is harder to mitigate, but some testing has empirically shown that the returned value is often the choice between `0` and `1`.

Another disadvantage is that it is local to the function the peripheral read occurs in. It does not look at what happens to the return value. This can cause wrong results when the return value of a function needs to be a specific value (other than `0` or `1`), and if it is not, an error state is entered.

Goal Tactic
This tactic tries to get to a specific address `x` and accepts all states that reach this address. In this sense, it is a generalisation of the Dummy Tactic.

The main advantage of this tactic is that this tactic has the potential to be less local than the Return Tactic, and thus is able to find better constraints. It also makes very few assumptions on the underlying instructions. The function that contains the peripheral read does not need to return, and it does not matter what calling convention it uses.

The main disadvantage is that this tactic can lead to state explosion if the goal address is sufficiently far away. This causes the symbolic execution to take a long time. This can be mitigated by using a timeout on the symbolic execution and then using a control flow graph-based distance metric to find the state that got the closest to the goal address `x`.

Another disadvantage is determining the goal address `x`. Although automated solutions to this might be possible, this work leaves determining the precise address to the (human) analyst. This makes this tactic unsuitable for being automatically selected, but an analyst can force the system to use this tactic when the other tactics struggle to find the right path. The analyst can simply direct the symbolic execution towards a specific instruction.

Steps Tactic
This tactic tries to get `n` basic blocks (steps) ahead, avoiding infinite loops and other error states.

The main advantage is that this limits the state explosion the Goal Tactic can cause, while still being less local than the Return Tactic. Like with the Goal Tactic, this will cause more accurate constraints, and makes fewer assumptions on the underlying instructions than the Return Tactic. Another advantage is that this tactic does not require a specific address, so it can be automatically selected.

The main disadvantage of this tactic is that it is guaranteed that the goal state is `n` steps away. Although the goal in the Goal Tactic can be quite close, thus requiring very little symbolic execution, this is never the case for the Steps Tactic.

Another disadvantage is that the parameter `n` needs to be determined. Other research ([4], [23]) employ similar techniques, where they set `n` to 3 and 5, respectively. Since 5 is still relatively small, and leads to more accurate results than 3, the tactic selection only uses the value 5 for `n`.

5 Implementation

In the previous chapter, the design decisions that were made to reach a final design of the system discussed in this thesis were described. In this chapter, we describe the final design (shown in Figure 5.1) and the various components that it consists of and their requirements. Section 5.1 describes the emulator component. Section 5.2 describes the analysis tool component. Section 5.3 describes the read resolver component, which is the layer between the concrete and symbolic execution. Section 5.4 describes the backtracker component, which resets the system when it has become stuck. Finally, in Section 5.5, we describe the history component, which keeps track of all the peripheral reads that occurred.

5.1. Emulator

The first component is the emulator. This component is responsible for executing the code, even if this code is not native to the machine the system is run on. Although this is a complex problem, solutions have been created already, such as QEMU [2] and Unicorn [41]. As such, we will not discuss the various challenges associated with this problem.

This component is required to be able to communicate with the Read Resolver component when an address belonging to a peripheral is read. It should also be able to activate the Backtracker component when a bad state has been detected.

5.2. Analysis Tool

This component is used to store persistent information to speed up future executions of the same binary. Its analysis features are also put to use to define the precise workings of some tactics. In Section 5.2.1, the precise usage of the analysis features is described, and in Section 5.2.2, a feature is described

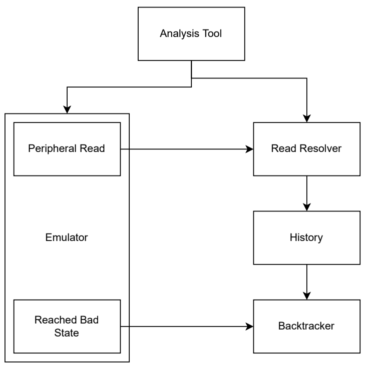


Figure 5.1: Diagram that shows how the 5 components of the proposed design relate to each other.

that allows human analysts to influence the tactic selection procedure to lead to fewer backtracking attempts.

5.2.1. Analysis Features

The analysis features of this tool are used by the Read Resolver component. An example of the information used by the Read Resolver is the address ranges that the instructions of a function comprise. This is used to map an address to the function that contains it. Another example is the (optional) annotations that a human analyst has made. These annotations are used to guide the Read Resolver into the right direction, and avoid backtracking.

5.2.2. Manual Annotations

The persistent information that is stored includes things like the known good values, or hints on the tactic. This is stored in such a way that it is easily editable by human analysts. This allows human guidance and enables fixing wrong values being resolved in case the Backtracking does not suffice.

These annotations are made using comments in Ghidra. When an “EOL” comment is used, it applies to the peripheral read that is made while the instruction at that specific address is executed. Alternatively, a “Plate” comment can be made at the first instruction of a function. The annotation made in this comment applies to all peripheral reads originating from that function. When there is an annotation in the “EOL” comment at a peripheral read in a function that also has an annotation in the “Plate” comment, the annotation made using the “EOL” comment takes precedence.

1. `return` causes a `ReturnTactic(None)`, that is, a `ReturnTactic` that accepts any return value.
2. `return == <n>` causes a `ReturnTactic(<n>)`, that is, a `ReturnTactic` that returns the value `<n>`.
3. `goto <a>` causes a `GoalTactic(<a>)`, that is, a tactic that tries to reach address `<a>`.
4. `step <n>` causes a `StepTactic(<n>)`, that is, a tactic that tries to reach `<n>` basic blocks ahead.

5.3. Read Resolver

This component responds to peripheral reads encountered by the Emulator. This component is novel and uses multiple steps to find a suitable value for the peripheral read. If the wrong value is selected, the Backtracker component will cause the peripheral read to be resolved again. The steps are as follows, and will be further discussed in the following subsections.

1. Select an appropriate tactic to define various properties of the symbolic execution.
2. Run the Symbolic Execution to obtain possible values.
3. Select one of the possible values.
4. Save the used tactic, selected value and the emulator state before this read in the history component.

Figure 5.2 shows the steps in a diagram. The ‘saving’ step is represented by the arrow going towards the History.

5.3.1. Tactic Selection

Having several different tactics with different trade-offs is nice, but we can only use one tactic at a time. This means we have to choose one of the tactics. Choosing the wrong tactic can lead to a bad state, which requires the use of the Backtracker and thus increases the time cost of the system. As such, choosing the correct tactic is an integral part of the proposed system. To this end, we use several different, hardcoded, tactics in order. Whenever a peripheral read happens, the first tactic on that list is used. When a backtracking attempt causes the same peripheral read to happen again, we ‘escalate’ to the next tactic on the list.

Despite the simplicity of this idea, its performance depends on the composition of the list. Ideally, the list should start with tactics that are relatively simple for the Symbolic Executor to solve. Then, as the list continues, the tactics should allow the Symbolic Executor to discover more complex constraints. The list is represented on the next page.

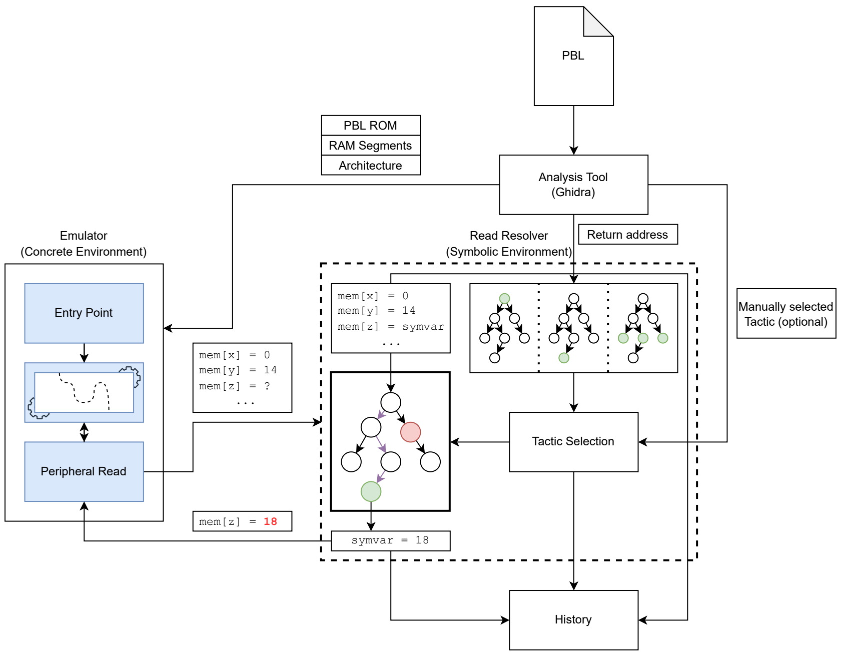


Figure 5.2: Diagram that shows how the Read Resolver component works.

1. `DummyTactic`
2. `ReturnTactic(0)`
3. `ReturnTactic(1)`
4. `StepsTactic(5)`
5. `StepsTactic(10)`

Some initial experimentation has shown that most peripheral reads do not influence control flow. An example is some flags on the peripheral being updated. In these cases, it does not matter what value is actually read. As such, we will choose the `DummyTactic` first to minimise the time spent in the Symbolic Executor.

If the `DummyTactic` leads to an error state, we will eventually escalate to the next tactic on the list, which is `ReturnTactic(0)`. We found that most functions will return the value 0 to indicate success. The specific constraint of returning 0 gives more direction to the Symbolic Executor, which therefore can discard more possible paths and give a result that is more tightly constrained.

However, sometimes, the function uses the value 1 to indicate successful execution. In this case, returning 0 will likely lead to an error state, causing escalation, and thus `ReturnTactic(1)` to be used. Its advantages and disadvantages are the same as `ReturnTactic(0)`.

Next, there is `StepsTactic(5)`. In Section 4.5, we have already motivated the choice for the parameter 5. The purpose of this tactic is to catch the cases where either `ReturnTactic` did not succeed. This can happen because the expected return value of the function containing the peripheral read is neither 0 nor 1, or because the error state was entered before the end of the function. This tactic serves to look across function boundaries, and avoid error states. This allows the Symbolic Executor to find more precise constraints on the value.

Finally, `StepsTactic(10)`. If looking 5 steps into the future misses an error state, it might be helpful to look 10 steps into the future. Although there might be cases where even this is not enough, in our testing, this seems sufficient.

In addition, an extra constraint is used to make sure the returned value does not equal any of the previously returned values (if there are any). Since the only output from the Read Resolver is a single value, returning the same value multiple times will result in the same path being walked, and thus the system inevitably ends up in the same error state as the previous execution with this return value.

Escalation
As mentioned before, escalation refers to changing the tactic that is used to a different tactic. This can occur in one of the following four scenarios:

1. **Backtracking**. An error state was reached during emulation, and the system has been reset to this peripheral read.
2. **Same read happens more than 10 times**. If the same peripheral address is read more than 10 times in a row from the same code address, this is indicative of the code checking the status of the peripheral continuously, presumably checking for a certain semaphore value. Additionally, the previously used tactic has not brought the system closer to exiting this loop, so another tactic should be used.

   To detect these cases, a sliding window with size for 10 elements is kept, with each entry consisting of a tuple containing the address of the program counter when the peripheral read happens, and the peripheral address that is read. If the sliding window is filled with identical tuples, the system escalates.
3. **The chosen tactic was not applicable**. Sometimes, a tactic is not applicable to a peripheral read. With the currently implementated tactics, this only happens when the `ReturnTactic` is chosen, and only when it is used for a peripheral read that occurs outside any function. This only happens in the firmware's entry point, which has no return instruction. Of course, if a tactic is not applicable, another tactic should be chosen.
4. **The chosen tactic did not give any results**. This case occurs when the constraint generated by the current tactic is unsatisfiable, that is, there is no value that can satisfy the constraint. An example of this is when a tactic attempts to read a value that has already been used before without success.

Manual override
Although the automatic tactic selection and backtracking mechanism should always find a value that avoids all error states, this can take a very long time in some cases, especially if the peripheral read and the decision to enter an (obvious) error state are many instructions and peripheral reads apart. In these cases, a human analyst can specify a specific tactic to be used for peripheral reads originating from a specific instruction, or from any instruction within a specific function.

This is specified in the Analysis Tool, using the feature to place comments at certain addresses. The tactic selector always loads the comments in the Analysis Tool at the address the read comes from and the entry point of the function containing that contains this address and tries to parse these comments. If any of the comments specify a tactic, this tactic is used initially instead of the `DummyTactic`. In case of conflicting tactics being specified at the instruction and at the entry point, the tactic specified at the instruction takes precedence. This is explained in more detail in **??**.

5.3.2. Symbolic Execution

The Symbolic Executor translates the machine instructions into constraints. Whenever there are multiple paths that can be taken, for example if there is a branch that depends on a symbolic variable, both paths are analysed / executed simultaneously. Tactics are used to configure the symbolic execution and tell the Symbolic Executor which states to concretise from. The Read Resolver selects a tactic and starts the symbolic execution using this tactic.

Although there exist several systems that can perform symbolic execution, this design uses `angr`, which in turn uses `z3` for the SMT solving. However, there is nothing that depends on `z3` specifically, so a different SMT solver could be used with minimal effort.

5.3.3. Value Selection

The Symbolic Executor typically provides several different values for the read that would lead to a state that satisfies the selected tactic. This part of the component selects one of these values. Since,

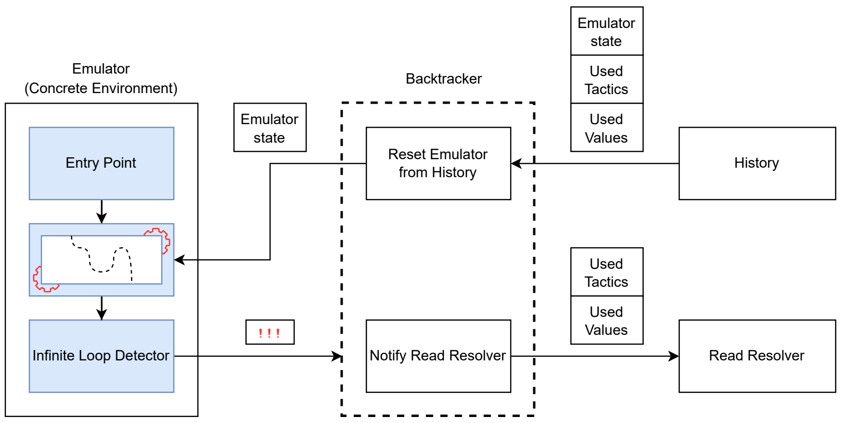


Figure 5.3: Diagram that shows how the Backtracker component works.

according to the Symbolic Executor, all values will reach the same state, this component does not do anything complicated, and just picks randomly from the possible values.

5.3.4. Saving

The final part of this component saves the state of the emulator, the chosen tactic and the chosen value to the History (see Section 5.5). This information can later be used by the backtracking system.

5.4. Backtracker

This component is responsible for backtracking when another component finds that the system has become stuck. This can be the Read Resolver that could not find any tactic that could lead to a possible value, or it could be the Emulator that detected that we ended up in a bad state.

The backtracker looks at the last saved entry to the history stack, and resets the emulator to the state specified by that entry. Next, it modifies the tactic selection to avoid the same tactic being selected, and it avoids this same value being returned.

5.5. History

This component stores the peripheral reads that have occurred in the past. This uses a simple stack (last-in-first-out) data structure. The Read Resolver pushes an entry to the stack every time a read happens, and the Backtracker pops an entry whenever a backtracking attempt is made. Each entry consists of three items: the emulator state at the start of a peripheral read, a list of tactics that have already been used for this peripheral read and finally a list of values that have already been attempted for this peripheral read.

6 Evaluation

The evaluation of the system described in this thesis consists of three parts. In Section 6.1, the accuracy of the emulation when using the system described in this thesis, by comparing the execution of several samples in the system and their execution on real hardware. The second part, described in Section 6.2, evaluates the process of tactic selection. It specifically looks at what combination of tactics performs best on several different samples. Finally, in Section 6.3, the system is compared against a similar system from the state of the art, Jetset [27].

6.1. Final State

To evaluate how well the trace and state generated by the system compare to traces and states that can be reached using real hardware, we used a development board to collect an execution trace and the state at a pre-defined point when running a binary that was specifically created for this evaluation. Then, we run the same binary in the system discussed in this thesis. Finally, we compare the artefacts that result from both approaches to evaluate how realistic the state produced by the system discussed in this thesis is.

6.1.1. Method

The high level idea behind this method is to collect data on how the real hardware executes some code that interfaces with peripherals, and compare that to the way this same code is executed in the system developed in this thesis.

The hardware that was used is the Raspberry Pi 4B. This choice is motivated by the wide availability, facilitating researchers wishing to reproduce the experiments conducted in this work. In addition, this hardware has a JTAG interface, which allows researchers to attach debug tools and follow the state of the device during execution.

The software that was used, is based on the various bare-metal programs that were published as a tutorial[1]. This choice was made because of the availability of the source code, and the gradually increasing complexity of the programs. Although these binaries were originally intended for the Raspberry Pi 3, they can still be used as a reference for developing similar code for the Raspberry Pi 4 because of the many similarities between the models. The main difference between the two models is that the peripherals are mapped to different addresses.

In contrast to related work [5], the similarity metric used is the Levenshtein Distance (see Section 2.5.2). This choice was made because the order in which code executes was considered important, and the similarity metric used in this related work does not take this into account. However, even this metric is not complete, as loops that perform no meaningful action other than waiting can be executed more or fewer times without having any effect on the state of the program at any point (disregarding the number of executed instructions). Such loops happen frequently in embedded software, and especially where

[1]Available for download at `https://github.com/bztsrc/raspi3-tutorial`

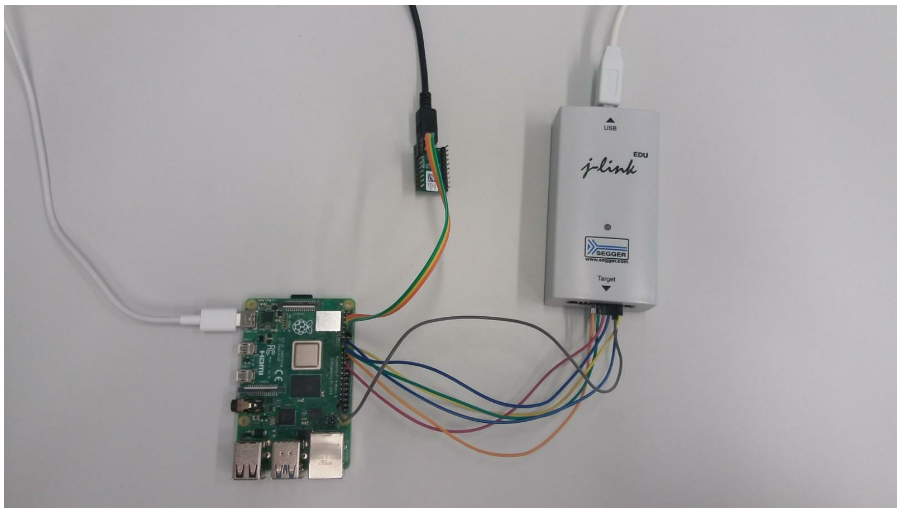


Figure 6.1: A photograph demonstrating the experimental setup, showing, from left to right, the Raspberry Pi 4B, the UART to USB bridge and the J-Link EDU. The computer is omitted from this picture for clarity.

such loops wait for peripherals, the system described in this thesis would use fewer iterations than real hardware.

We used JTAG because it can read data and interact with the processor at the lowest level. This allows determining the state of the processor after every instruction, which enables collecting fine-grained results.

We chose to use the Raspberry Pi 4, Model B, with 8 GB of RAM. However, none of the experiments used require this much RAM, and should be reproducible on versions of the Raspberry Pi 4, Model B, with less RAM. This choice was largely motivated by the availability of Raspberry Pi boards. These boards are widely sold, with models starting at 35 USD.

6.1.2. Setup

Board

Apart from the board itself, a J-Link EDU was used to control the execution using JTAG. In addition, the serial output pins were also connected to a laptop, which was used to monitor the device. A photograph of the experimental setup is shown in Figure 6.1.

After the components are connected as shown in Figure 6.1, an SD card needs to be prepared. This SD card will contain the various kernel images that are tested. This preparation consists of the following steps:

1. Use the Raspberry Pi Imager[2] on a micro SD card. This reformats the SD card and creates two partitions, named `boot` and `rootfs`. Only the `boot` partition will be used while conducting these experiments.
2. Change the `config.txt` in the `boot` partition of the SD card. A few extra settings are required to enable UART and JTAG output to interact with the examples and collect the traces. These settings are as follows:

```
[pi4]
enable_uart=1
enable_jtag_gpio=1
kernel=kernel8.img
```

3. Remove or rename all files in the `boot` partition of the SD card whose name starts with `kernel`. This ensures that Raspberry Pi OS is not booted, and instead our example is. For each of the examples, a `kernel8.img` file will be generated, which will replace the default files.

[2]Available for download at `https://www.raspberrypi.com/software`

Whenever a new sample needs to be tested, the Makefile will ensure a new kernel8.img file is generated, which should be placed in the boot partition of the SD card.

Computer
In addition to the Raspberry Pi, another computer was used to communicate with the Raspberry Pi and compile the source code of the experiments. This computer runs Ubuntu 20.04 LTS as an operating system.

Since the Raspberry Pi, unlike the development computer, runs on an ARM Cortex-A72 processor, a cross-compiler is required to compile the examples for the Raspberry Pi. The compiler that was used is gcc version 8.1.0, which was custom-built for the aarch64-elf target. The commands required for installing this are given in Section 6.1.3. Furthermore, make is required to build the different kernels and can be installed using sudo apt-get install make. Additionally, OpenOCD is required to communicate with the J-Link and connect a debugger to the Raspberry Pi. The version of OpenOCD that was used was a custom build of release version 0.11[3]. The debugger that was used is gdb-multiarch (9.1-0ubuntu1). The debugger can be installed by simply running sudo apt-get install gdb-multiarch. Furthermore, Ghidra[4] is required. This experiment was conducted using version 10.1.2. However, later versions of Ghidra are likely to also work. Next, the Ghidra Bridge extension[5] (version 0.2.5) is required to allow communication between Ghidra and the emulator.

6.1.3. Process

This subsection describes what steps are required to run the experiments on the Raspberry Pi.

1. First, the experiments have been published as a GitHub repository[6]. This repository contains all code required, as well as collected data. To reproduce these results, it is important to clone this repository.
2. Next, the necessary dependencies should be setup, as mentioned in Section 6.1.2. The following bash commands can be used to quickly install the required dependencies:

```
$ sudo apt-get install gdb-multiarch
$ wget https://ftpmirror.gnu.org/binutils/binutils-2.30.tar.gz \
    https://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz \
    https://gcc.gnu.org/pub/gcc/infrastructure/isl-0.18.tar.bz2 \
    https://ftpmirror.gnu.org/gcc/gcc-8.1.0/gcc-8.1.0.tar.gz \
    https://ftpmirror.gnu.org/mpfr/mpfr-4.0.1.tar.gz \
    https://ftpmirror.gnu.org/gmp/gmp-6.1.2.tar.bz2 \
    https://ftpmirror.gnu.org/mpc/mpc-1.1.0.tar.gz
$ for i in *.tar.gz; do tar -xzf $i; done
$ for i in *.tar.bz2; do tar -xjf $i; done
$ rm -f *.tar.*
$ cd binutils-*
$ ln -s ../isl-* isl
$ cd ../gcc-*
$ ln -s ../isl-* isl
$ ln -s ../mpfr-* mpfr
$ ln -s ../gmp-* gmp
$ ln -s ../mpc-* mpc
$ ln -s ../cloog-* cloog
$ cd ..
$ mkdir aarch64-binutils
$ cd aarch64-binutils
$ ../binutils-*/configure --prefix=/usr/local/cross-compiler --enable-plugin \
    --target=aarch64-elf --enable-shared --enable-threads=posix --enable-libmpx\
```

[3]Available for download at https://github.com/OpenOCD-org/OpenOCD/releases/tag/v0.11.0
[4]Available for download at https://github.com/NationalSecurityAgency/ghidra/releases/tag/Ghidra_10.1.2_build
[5]Available for download at https://github.com/justfoxing/ghidra_bridge/releases/tag/0.2.5
[6]Available for download at https://github.com/LukeSerne/ConFEDSS-evaluation

```
    --disable-libunwind-exceptions --enable-clocale=gnu --disable-libstdcxx-pch\
    --with-linker-hash-style=gnu --with-gnu-ld --enable-gnu-indirect-function \
    --disable-linker-build-id --disable-multilib --enable-install-libiberty \
    --enable-default-pie --enable-default-ssp --enable-gnu-unique-object \
    --with-system-zlib --enable-__cxa_atexit --with-isl --disable-libssp \
    --disable-werror --enable-lto --enable-checking=release
$ make -j4
$ sudo make install
$ cd ..
$ mkdir aarch64-gcc
$ cd aarch64-gcc
$ ../gcc-*/configure --prefix=/usr/local/cross-compiler --target=aarch64-elf \
    --enable-languages=c --enable-shared --enable-threads=posix --enable-libmpx\
    --disable-libunwind-exceptions --enable-clocale=gnu --disable-libstdcxx-pch\
    --disable-libssp --disable-linker-build-id --enable-default-ssp --with-isl \
    --enable-checking=release --enable-default-pie --enable-gnu-unique-object \
    --with-system-zlib --enable-__cxa_atexit --enable-plugin --enable-lto \
    --enable-install-libiberty --with-linker-hash-style=gnu --with-gnu-ld \
    --enable-gnu-indirect-function --disable-multilib --disable-werror
$ make -j4 all-gcc
$ sudo make install-gcc
```

3. Now everything is installed, the Raspberry Pi should be connected to the computer. To this end, Figure 6.1 can be used for reference, where the UART to USB bridge and J-Link should be connected to the USB ports of the computer. Additionally, there are various sources available online to assist in the wiring.[7][8]
4. Choose an experiment to run. Each of the experiments corresponds to a value from `01` through `04` that will be referred to as the `<EXPERIMENT_ID>`.
5. Compile the kernel corresponding to the chosen experiment by running `make` in the correct folder. The source code for the experiments is located in the repository, in the `code` subfolder. Every experiment has its own source code, which is located in different folders. Each of these folders is prefixed by the `<EXPERIMENT_ID>`, so with the help of tab-completion, it should be straight-forward to change to the correct directory.

   ```
   cd /path/to/experiments/code/<EXPERIMENT_ID>
   make
   ```
6. Copy the `kernel8.img` file that was created in the `out` folder to the `boot` partition of the SD card.
7. Unmount the SD card from the computer and insert it into the Raspberry Pi and connect the Raspberry Pi to power.
8. On the computer, make sure the computer is connected to the J-Link through a USB port and connect to the Raspberry Pi with OpenOCD using the command below. The required configuration file is included in the repository.

   ```
   sudo openocd -f /path/to/experiments/configs/rpi4.cfg
   ```

   If everything is connected properly, OpenOCD should, after a short delay, print something similar to `Info : Listening on port 3333 for gdb connections`. If not, simply repeat the previous step until it works, and ensure all wires are connected properly.
9. Connect `gdb` to OpenOCD by running the `gdb-multiarch` command below. Inside `gdb`, run the following command to connect to OpenOCD. Make sure to connect to the correct port, which is `3333` by default.

   ```
   target extended-remote :3333
   ```

[7]A clear diagram of the pinout of the Raspberry Pi 4B can be found at https://pinout.xyz.

[8]The pinout of the J-Link EDU can be found at https://www.segger.com/products/debug-probes/j-link/technology/interface-description.

`<EXPERIMENT_ID>`	`<END_ADDRESS>`	`<LOGGING_FILE>`
01	0x80368	/data/trace_01.txt
02	0x80390	/data/trace_02.txt
03	0x80b40	/data/trace_03.txt
04	0x8078c	/data/trace_04.txt

Table 6.1: Experiment parameters for different experiments. `<END_ADDRESS>` refers to the address of the last instruction that is executed. `<LOGGING_FILE>` is the path where the corresponding default trace can be found in the provided repository.

10. The Raspberry Pi is probably stuck in the initial loop. To break out of the loop, the counter register that is used (`x0`) can be set to 1 to exit the loop either in the current iteration or the next iteration. This can be done using the `gdb` command below.

    ```
    set $x0 = 1
    ```

11. The following `gdb` commands have been used to collect the data. The 'variable's `<LOGGING_FILE>` and `<END_ADDRESS>` need to manually be replaced by the filename where the trace is to be saved (for example `example_trace.log`) and the address of the first instruction of the infinite loop that is part of the echo mode. The `<END_ADDRESS>` and `<LOGGING_FILE>` corresponding to the current `<EXPERIMENT_ID>` can be looked up using Table 6.1.

    The `gdb` commands that have been used for data collection are:

    ```
    set height 0
    set logging file <LOGGING_FILE>
    set logging on
    while ($pc != <END_ADDRESS>)
    si
    end
    i r
    x/1024x 0x7f000
    set logging off
    ```

12. After that has finished running, the log file should have been written to `<LOGGING_FILE>`. Power can now be removed from the Raspberry Pi, and OpenOCD and GDB can be closed. Unfortunately, the log file generated by GDB is not very clean, because OpenOCD interjected various debugging messages. However, this can be cleaned up by using find and replace to remove the extra data below:

    ```
     in ?? ()
    bcm2711.a72.1 halted in AArch64 state due to debug-request, current mode: EL2H
    cpsr: 0x000003c9 pc: 0x80
    MMU: disabled, D-Cache: disabled, I-Cache: disabled
    bcm2711.a72.2 halted in AArch64 state due to debug-request, current mode: EL2H
    cpsr: 0x000003c9 pc: 0x80
    MMU: disabled, D-Cache: disabled, I-Cache: disabled
    bcm2711.a72.3 halted in AArch64 state due to debug-request, current mode: EL2H
    cpsr: 0x000003c9 pc: 0x80
    MMU: disabled, D-Cache: disabled, I-Cache: disabled
    ```

13. Now open Ghidra and load the `kernel8.img` file that was built into the `out` folder inside the `code` folder containing the source code for the experiment. The base address should be `0x80000` and the language `AARCH64:LE:64:v8A` (version 1.5).
14. In Ghidra, open the "Script Manager" dialogue and run the `ghidra_bridge_server_background.py` script. This dialogue is accessed through the "Window" menu item.
15. Use the following command to start the emulator with the correct hooks file. The exact hooks file that is necessary depends on the example that is loaded. However, these hooks files are very basic and only set up the symbolic fallback for peripheral communication and intercept UART

messages. The hooks scripts are prefixed by a number (`01` through `04`). As such, tab-completion can be used to easily find the correct full path.

```
python firmulator.py --hooks /path/to/experiments/hooks/<EXPERIMENT_ID>
```

16. The logs of the system should be in the file `/experiments/data/sym_<EXPERIMENT_ID>`. A file highlighting the difference between the two files can be obtained using any of a number of tools that have been developed to compare text files. In this process, `git`'s `diff` feature was used.

```
git diff /path/to/experiments/data/trace_<EXPERIMENT_ID> \
    /path/to/experiments/data/sym_<EXPERIMENT_ID> \
    > /path/to/experiments/data/<EXPERIMENT_ID>.diff
```

Afterwards, various simple analyses were performed on the resulting diff file, such as counting the number of entries, which corresponds to the Levenshtein distance.

Steps 4-16 should be repeated for each of the four examples. Afterwards, 4 files should have been produced. These 4 files indicating the differences are analysed in Section 6.1.5.

6.1.4. Examples

Four different example kernel programs were developed. Each of these programs uses a different form of interaction with peripherals, and is designed to test the strengths and limitations of the system that is described in this thesis.

To simplify collecting complete and consistent traces, all examples started by incrementing a 32-bit counter until it overflowed[9]. This gives the JTAG interface time to set up and ensures all traces start from the same point. When GDB is attached to the device, the counter can be set to a much higher value, just before the overflow to reduce excessive waiting.

UART / Mailboxes

This example sends some messages over the UART output of the Raspberry Pi, and also waits for some input. Additionally, this example uses the mailboxes, which is a communication channel between the CPU and GPU. This channel is used to transmit information about several aspects of the device such as the device's serial number, but also the screen resolution and more. In this example however, we only read the device serial number. Then, the serial number is sent over UART, along with some textual messages.

The Raspberry Pi interfaces with the UART peripheral by reading from and writing data to memory segment belonging to the peripheral (using memory-mapped IO). The mailboxes interface uses a global variable containing and array of 144 bytes. The CPU chooses an address for this array of bytes and communicates this to the GPU using memory-mapped IO.

A snippet of the `main` function is shown in Figure 6.2.

[9]Although this counter increases in the source code, the compiler changes this loop into decreasing a counter until it reaches 0. That is also why step 10 in Section 6.1.3 sets the counter to 1 instead of 0xFFFFFFFF.

```c
uart_init();

uart_puts("Hello, world!\n");

uint64_t serial = 0;
uint8_t error_getting_serial = get_board_serial(&serial);
if (error_getting_serial) {
    uart_puts("Unable to query serial!\n");
} else {
    uart_puts("My serial number is: ");
    uart_hex(serial >> 32);
    uart_hex(serial & 0xFFFFFFFF);
    uart_puts("\n");
}
```

Figure 6.2: Snippet of the source code of the UART / Mailboxes experiment.

Random / Timer

This example initialises the peripheral responsible for random number generation and reads several random values and reports those over UART. In addition, the system timer is used several times to implement a function that waits a specific number of milliseconds. Both the RNG peripheral and the system timer are interfaced with using memory-mapped IO.

A snippet of the `main` function is shown in Figure 6.3.

```c
uart_init();
rand_init();

uart_puts("The random number was chosen to be: ");
uart_hex(rand(0, 0xFFFFFFFF));
uart_puts("\n");

uart_puts("Waiting 1000000 usec (using timer)... ");
wait_usec(1000000);
uart_puts("DONE\n");
```

Figure 6.3: Snippet of the source code of the Random / Timer experiment.

Framebuffer

This program shows a small image on the screen that is connected to the Raspberry Pi through the micro-HDMI port. The size of this image is chosen to be quite small to make the process of collecting the trace on the Raspberry Pi faster. During the collection of the trace, the device ran very slowly. This caused it to only be able to write a handful of pixels every second. Since there is no meaningful difference in behaviour between writing the first and the millionth pixel, a choice was made to reduce to total number of pixels, to reduce the time needed to collect the trace. A final size of 10x10 pixels was chosen. Additionally, a string of characters is written to the screen.

A snippet of the `main` function is shown in Figure 6.4.

```c
// set up serial console and linear frame buffer
uart_init();
lfb_init();

// display a pixmap
lfb_showpicture();

// display an ASCII string on screen with PSF
lfb_print(80, 80, "Hello World!");

// display a UTF-8 string on screen with SSFN
lfb_proprint(80, 120, "Hello többnyelvű World!");
```

Figure 6.4: Snippet of the source code of the Framebuffer experiment.

ELF Parser
This program expects an ELF file as input over UART. It then parses the ELF header and performs some checks to ensure that the provided ELF file has a proper header and can be run on the Raspberry Pi 4. If a valid ELF file is provided, a response is sent back over UART, giving some information about the ELF file.

While collecting the trace for this program on the real hardware, there were some issues. These issues are caused by the reduced speed of the processor in the trace collection setup, as no issues were found when running the program without collecting a trace. The issue that occurred while tracing the execution was that the input of the 9th byte of input would fail. This was solved in the following way: First, a trace was collected up to and including the first few bytes of input. Then, a breakpoint would be placed after the input loop function and no trace would be collected until the breakpoint was hit. After the breakpoint was hit, the rest of the program's execution would be traced again.

Although concatenating these two partial traces does not give the complete trace, it is possible to reconstruct the complete trace. That is because the loop that reads input contains no conditional branches and thus has the same instruction pattern on every iteration. Furthermore, the number of iterations is fixed and the same between each run. As such, we can isolate the trace of one iteration of the input loop from the first partial trace, duplicate that as many times as necessary to have a final trace that executed the loop the correct number of iterations. This infix is then used to join the two partial traces to obtain a complete trace.

A snippet of the `parse_elf` function that checks the received ELF file's header is shown in Figure 6.5.

```c
// Check that this is an Aarch64 ELF file
if ((elf_hdr->e_machine != EM_AARCH64) || (elf_hdr->e_version != EV_CURRENT)) {
    return 0;
}
// Check that the entry point is in the correct range
if (! ((0x00010000 < elf_hdr->e_entry) && (elf_hdr->e_entry < 0x00100000))) {
    return 0;
}
```

Figure 6.5: Snippet of the source code of the ELF Parser experiment.

6.1.5. Results
The results that have been collected have been summarised in Table 6.2. "HW" refers to the execution of the experiment on real hardware, whereas "Emu" refers to the execution of the experiment within the system described in this thesis.

Metric	Experiment			
	1	2	3	4
Number of peripheral reads	12	15	20	124
HW: Num. instructions	1696	2317	2197	18938
HW: Time (seconds)	0.459	1.487	1.375	2.812
Emu: Num. instructions	1170	1263	1793	3484
Emu: Time (seconds)	32.4	40.5	62.4	213.8
Levenshtein distance (insn)	526	1054	404	15454
Levenshtein distance (data)	3	0	0	0

Table 6.2: Results for Final State comparison.

6.2. Tactic Selection

This part of the evaluation concerns the automatic tactic selection system. To answer the research question about the best way to automatically handle peripheral communication, we test the different combinations of tactics. We measure their performance based on two metrics: the number of peripheral reads and the time it takes to reach the end of a program. The number of peripheral reads can indicate the different paths taken by different combinations of tactics, but it is also relevant to the performance. This is because every peripheral read causes the execution to exit the emulator and call a function in Python. This can be considered a limitation of this implementation, rather than the method itself. When it comes to the total time required to fully execute the program, less time required is considered better.

We tested all possible combinations of the tactics suitable for automatic use, namely the `DummyTactic`, `StepsTactic(5)`, `StepsTactic(10)`, `ReturnTactic(0)` and `ReturnTactic(1)`. The precise definitions of these tactics are described in Section 4.5. These tactics were tested on various different samples to ensure generality of the results.

6.2.1. Method

The same program is ran multiple times, with a different combination of tactics to allow for easy comparison of the results of the different combinations of tactics. Since we have 5 tactics, 32 different runs are required to test all combinations of tactics for each sample. Additionally, we ran the same four different programs that were used in Section 6.1 to completion to ensure the results would be more broadly applicable. As the intricacies of those examples were already described in detail in Section 6.1.4, we refer the reader to that section for more information on the nature of these samples. The metrics that will be used are the number of total peripheral reads and the time in seconds that was used to reach the endpoint.

6.2.2. Setup

The data required for the experiments has been published as a GitHub repository[10]. For this experiment, it contains the hook scripts required for the successful execution. Furthermore, Ghidra[11] is required. This experiment was conducted using version 10.1.2. However, later versions of Ghidra are likely to also work. Next, the Ghidra Bridge extension[12] is required to allow communication between Ghidra and the emulator.

6.2.3. Process

1. The first step is to load one of the samples into Ghidra. It is assumed that the samples have already been compiled for 6.1. If not, the process of compiling the samples is described there.

   Open Ghidra and load the `kernel8.img` file that was built into the `out` folder inside the `code` folder containing the source code for the selected experiment. The base address should be `0x80000` and the language `AARCH64:LE:64:v8A` (version 1.5).

[10]Available for download at `https://github.com/LukeSerne/ConFEDSS-evaluation`
[11]Available for download at `https://github.com/NationalSecurityAgency/ghidra/releases/tag/Ghidra_10.1.2_build`
[12]Available for download at `https://github.com/justfoxing/ghidra_bridge/releases/tag/0.2.5`

2. In Ghidra, open the Script Manager and start the "Ghidra Bridge Background" script.
3. Modify the list of possible tactics that are used when a peripheral read occurs (see Section 5.3.1). This list is defined in the file `hooks/symbolic.py` at line 249:

```
249 # The possible tactics that can be employed. Every read should be able
250 # to be solved with a tactic from this list.
251 # <...>
252 candidates = [
253     DummyTactic(),
254     StepsTactic(max_depth = 5),
255     StepsTactic(max_depth = 10),
256     ReturnTactic(value = 0),
257     ReturnTactic(value = 1),
258 ]
```

   Individual elements of this list can be disabled by commenting them out using a # character at the beginning of the line. Other code relies on the relative order of the different tactics, so they should not be reordered.
4. Start the firmulator with a specific hooks script that logs instructions and breaks at the end point. The exact script path that is required is provided in the repository. Because the name of the required hook script starts with the `<EXPERIMENT_ID>`, tab-completion can be used to easily find the exact path.

```
python3 /path/to/firmulator.py --hooks /path/to/experiments/hooks/<EXPERIMENT\_ID>
```

5. Wait until the firmulator has reached the end point. Once this happens, the number of peripheral reads and the time that was taken is printed to the terminal.
6. Go back to step 3 and pick a different combination of tactics, until all combinations have been used and statistics for them are available.
7. When all combinations have been used, a different experiment should be selected and the entire process should be repeated.

6.2.4. Results

The results of these experiments are represented in Table 6.3.

Used Tactic(s)					Experiment 1		Experiment 2		Experiment 3		Experiment 4	
Dummy	Steps		Return		Num. reads	Time (s.)	Num. reads	Time (s.)	Num. reads	Time (s.)	Num. reads	Time (s.)
	5	10	0	1								
○	○	○	○	○	—	error	—	error	—	error	—	error
●	○	○	○	○	—	timeout	—	timeout	—	timeout	—	timeout
○	●	○	○	○	9	17.723	21	30.076	14	29.277	180	113.678
●	●	○	○	○	10	17.225	21	29.912	15	30.467	192	115.199
○	○	●	○	○	9	17.532	21	29.634	14	29.525	180	113.737
●	○	●	○	○	10	16.964	21	29.076	15	31.735	192	115.440
○	●	●	○	○	9	18.039	21	28.546	14	29.878	180	114.084
●	●	●	○	○	10	17.353	21	28.746	15	31.692	192	115.656
○	○	○	●	○	—	error	—	error	—	error	—	error
●	○	○	●	○	10	17.562	21	29.890	—	timeout	—	timeout
○	●	○	●	○	9	19.726	21	29.215	14	28.689	180	113.569
●	●	○	●	○	10	16.418	21	28.347	15	29.203	192	120.844
○	○	●	●	○	9	18.208	21	29.279	14	30.047	180	114.062
●	○	●	●	○	10	17.954	21	29.930	16	33.486	192	115.984
○	●	●	●	○	9	17.541	21	29.882	14	29.620	180	113.110
●	●	●	●	○	10	17.372	21	29.959	16	35.767	192	116.931
○	○	○	○	●	—	error	—	error	—	error	—	error
●	○	○	○	●	10	18.996	21	29.292	—	timeout	—	timeout
○	●	○	○	●	9	21.228	21	30.043	14	29.711	180	113.973
●	●	○	○	●	10	20.289	21	30.123	16	32.334	192	115.944
○	○	●	○	●	9	18.171	21	29.282	14	30.583	180	114.457
●	○	●	○	●	10	17.916	21	29.629	16	32.287	192	116.634
○	●	●	○	●	9	18.536	21	30.379	14	29.272	180	113.864
●	●	●	○	●	10	17.894	21	30.492	16	37.301	192	116.417
○	○	○	●	●	—	error	—	error	—	error	—	error
●	○	○	●	●	10	23.359	21	30.477	—	timeout	204	132.375
○	●	○	●	●	9	18.427	21	31.203	14	29.576	180	114.905
●	●	○	●	●	10	18.067	21	30.690	16	35.195	192	115.760
○	○	●	●	●	9	17.564	21	29.969	14	30.155	180	114.812
●	○	●	●	●	10	17.074	21	30.425	16	34.217	192	116.738
○	●	●	●	●	9	21.944	21	30.796	14	29.561	180	113.943
●	●	●	●	●	10	20.847	21	31.612	16	40.427	192	124.007

Table 6.3: Overview of how well different combinations of tactics perform on several samples.

6.3. Comparison to Jetset

Johnson et al. [27] published Jetset, which aims to solve the same problem of emulating systems with unknown peripherals. Their approach is based purely on symbolic execution, and represents the state of the art when it comes to using symbolic execution to emulate systems with unknown peripherals. As such, this work is compared to Jetset to assess how well it improves the state of the art.

6.3.1. Method

This part of the evaluation has two components. The first component consists of running the examples that were used in Section 6.1 and Section 6.2 in Jetset. As the intricacies of those examples were already described in detail in Section 6.1.4, we refer the reader to that section for more information on the nature of these samples.

The second component is running the programs that were used to evaluate Jetset in Johnson et al. [27] in the system implemented in this work. However, of the four samples that were used in said paper, only two have been publicly released[13]. As such, this component is limited to only those two samples.

6.3.2. Setup

This experiment was conducted on the same computer that was used for the other parts of the evaluation. However, due to apparent compatibility issues of Jetset with Ubuntu 20.04 LTS, a virtual machine with Ubuntu 18.04 LTS was used.

Download Jetset[14] and install the various dependencies that are not installed on Ubuntu 18.04 LTS by default. Once the dependencies are installed, the `clone`, `config_qemu`, `build_qemu`, `virtualenv` and `build_jetset_engine` build targets will configure and build the various parts that are necessary to run the PBL in Jetset. This entire process can be performed by the following bash commands:

```
sudo apt-get install git make build-essential zlib1g-dev pkg-config libglib2.0-dev \
    binutils-dev libboost-all-dev autoconf libtool libssl-devlibpixman-1-dev \
    virtualenv xterm
make clone config_qemu build_qemu virtualenv build_jetset_engine
```

6.3.3. Process

For a more in-depth description of how to replicate the findings published by Johnson et al. [27], we refer the reader to the original paper. The results of running the samples in the system described in this work is given in Section 6.1.

Running samples in Jetset

This part describes what steps are required to run the samples provided in this work in the Jetset system.

1. First, activate the virtual environment for Jetset by executing the following command, and go to the directory that contains the driver script included with the Jetset repository.

   ```
   source jetset_env/bin/activate
   cd $(ENGINE_BASE)
   ```

2. Next, for each of the 4 different examples, use the following command to copy the Jetset configuration device model for the corresponding sample. Use a value of `01`, `02`, `03` or `04` for the `<EXPERIMENT_ID>` variable.

   ```
   cp /path/to/experiments/configs/<EXPERIMENT_ID>_config.py /jetset_engine/configs
   ```

3. Next, the new configuration should be added to `/jetset_engine/configs/socs.py`. On line 16 of this file, a dictionary of SoCs is defined. A new SoC needs to be added. These are defined by the `<EXPERIMENT_ID>_config.py` script that the previous step added to this folder. As such, this config file needs to be imported and added to this dictionary, as shown below. Several irrelevant parts of the code were replaced by `# ...` for clarity.

[13]Available for download at `https://github.com/aerosec/jetset_public_data`
[14]Available for download at `https://github.com/aerosec/jetset`

```
# ...
import configs.rpi_<EXPERIMENT_ID> as rpi_<EXPERIMENT_ID>

socs = {
    "cmu" : cmu,
    "beagle" : beagle,
    # ...
    "rpi_<EXPERIMENT_ID>": rpi_<EXPERIMENT_ID>,
}
# ...
```

4. Run the Jetset engine using the following command:

```
python $(ENGINE_SCRIPT) --soc=rpi_<EXPERIMENT_ID> --useFinalizer --useSlicer \
    --cmdfile /path/to/evalutation/scripts/run_jetset_<EXPERIMENT_ID>*.sh
```

5. After Jetset has finished, the number of peripheral reads that have been processed are printed to the console, as well as the total number of instructions executed and some part of the memory (the values at addresses `0x7f000` through `0x80000`) and the registers. This output should be saved to a file. In this case, this file will be named `/path/to/experiments/data/jetset_<EXPERIMENT_ID>`
6. The final step is comparing the results with the data obtained on the real hardware. Since the process of acquiring this data is described in Section 6.1, it will not be repeated here.

   A file highlighting the difference between the execution of real hardware and Jetset can be obtained using any of a number of tools that have been developed to compare text files. In this process, `git`'s `diff` feature was used.

```
git diff /path/to/experiments/data/trace_<EXPERIMENT_ID> \
    /path/to/experiments/data/jetset_<EXPERIMENT_ID> \
    > /path/to/experiments/data/<EXPERIMENT_ID>.diff
```

   Afterwards, various simple analyses were performed on the resulting diff file, such as counting the number of entries, which corresponds to the Levenshtein distance.

Running Jetset samples

1. First, the samples need to be downloaded. Only the Raspberry Pi 2 and Beaglebone-xM samples have been publicly released by Johnson et al., and as such, those will be used.[15]
2. The next step is to load these samples into Ghidra. The specific files that need to be loaded are `rpi/final/zImage` for the Raspberry Pi, which is an ELF file. Loading the Beaglebone-xM image can be done by loading the file `beagle/kernel/MLO`. This is a binary file that should be loaded at offset `0x402007F8`.
3. For the Raspberry Pi, the entry point is at the `stext` label, which is at address `0x80008000`. The goal is the `run_init_process` function, which is located at address `0x80102e04`. For the Beaglebone-xM, the entry point is at address `0x80102e04`, and the goal is located at the address `0x80008000`.
4. When the goal address has been reached, the newly generated log file contains an entry for every peripheral read and write, with their address. The time that was taken is printed to the terminal.
5. Finally, a file highlighting the difference between the two files can be obtained using any of a number of tools that have been developed to compare text files. In this process, `git`'s `diff` feature was used.

6.3.4. Results

The results for comparing the execution of samples have been summarised in Table 6.4. Table 6.5 contains the results of running the samples used by Johnson et al. [27] in the system described in this work.

[15]Available for download at `https://github.com/aerosec/jetset_public_data`

Metric	**Experiment**			
	1	**2**	**3**	**4**
Number of peripheral reads	12	15	20	124
Time (seconds)	40.4	56.1	74.3	224.8
Number of instructions	1170	1263	1793	3484
Levenshtein distance (insn)	0	0	0	0
Levenshtein distance (data)	3	0	0	0

Table 6.4: Results for Jetset comparison. The “Levenshtein distance (data)” was calculated by comparing the data of the final state of Jetset and the system described in this thesis.

	Raspberry Pi 2		**BeagleBoard-xM**	
CPU/SoC	Broadcom BCM2836		TI DM3730	
OS/SW	Linux 4.19.y		X-Loader	
	Jetset (Johnson et al.)	This Work	Jetset	This Work
Wall-clock time	6m54.16s	64m18.492s	5m20.741s	34m56.405s
MMIO writes	84,060	83,409	938	938
MMIO reads	83,857	83,324	3,633	3,623
MMIO write addrs	40	40	244	244
MMIO read addrs	37	37	61	61
Devices accessed	6	6	11	11

Table 6.5: Various metrics for the execution of samples used by Jetset in the system described in this work. Results for Jetset were taken from Table 1 in Johnson et al. [27]. For the wall-clock time, the sum of the three stages of Jetset was used.

7 Conclusion and Discussion

This chapter contains the Conclusion, Discussion as well as directions for Future Work. In the Conclusion section, Section 7.1, the initial research questions are answered and the results obtained in Chapter 6 are analysed. In the Discussion section, Section 7.2, the approaches of the research in this thesis are evaluated critically and the limitations of the work in this thesis are highlighted. Finally, in the Future Work section, Section 7.3, the limitations are discussed further and directions are given for future work to mitigate or remove these limitations.

7.1. Conclusion

First, we will answer the three research questions posed in Section 1.3.

1. **How can peripheral communication be handled automatically?**
   This question was explored by comparing existing solutions. Because of the scalability to new system and the possibility to produce realistic communication, an approach using symbolic abstractions of peripheral communication was chosen. This choice is further motivated in Chapter 3.
2. **How can peripheral communication be handled accurately?**
   This question was explored by collecting both the memory state that was reached by this system, and the path of instructions that was taken to get to this state. Then, the path and memory state were collected on real hardware and these two were compared. Section 6.1 describes the experimental setup, and Section 7.1.2 analyses the result and answers this question in greater detail.
3. **How can good and bad states be distinguished?**
   This question was explored using different definitions of good and bad states. By then comparing these different definitions (called tactics), conclusions can be drawn about the quality of different definitions. Section 6.2 describes the experimental setup, and Section 7.1.1 analyses the result and answers this question in greater detail.

Additionally, the system designed in this thesis is compared against the state of the art, which is described in Section 6.3 and analysed in Section 7.1.3.

7.1.1. Tactics

From Table 6.3, several conclusions can be drawn about the tactic selection procedure. First, at least a single tactic is required to avoid getting stuck or encountering an error, caused by trying to return from a non-returning function. The only tactic that can be used on its own is `StepsTactic(5)`.

Another interesting observation is that only two distinct paths were taken through the code; one with 30 peripheral reads and one with 32. The path with fewer peripheral reads is taken if and only if no `DummyTactic` is used. This indicates that the `DummyTactic` leads to more peripheral reads. This makes sense, because `StepsTactic` looks ahead and tends to prefer simpler paths, with fewer peripheral reads. Additionally, it is always faster to include a `DummyTactic`, even if it requires more peripheral reads.

Exclusively using the `ReturnTactic`s and `DummyTactic` does not work. This is because some peripheral reads occur inside functions that do not return. This means that the symbolic execution engine gets stuck when trying to find a path to the return instruction, since there is no such path.

Surprisingly, the fastest list of tactics is not the combination of all tactics, but instead consists of the `DummyTactic` and the two `StepTactic`s. It also does not use any `ReturnTactic`. The reason why more tactics are not necessarily better, is because the more tactics are available, the longer it takes for the system to start backtrack from a wrong value, since all options for tactics need to be exhausted before backtracking is initiated. If there are more options, it naturally takes longer before backtracking starts.

7.1.2. Final State

As can be seen from Table 6.2, the emulator is able to drastically reduce the number of executed instructions. As a result, the instruction path is quite different between the real hardware and the emulator. However, closer inspection reveals that the only difference in the traces consists of loops that are exited early by the emulator. The emulator is able to do this because these loops iterate until a peripheral sets a flag. Although this may take many iterations in reality, the emulator detects that this flag is a peripheral read. Because the emulator wants to avoid being stuck in the loop, a `StepsTactic` is used to symbolically evaluate the loop guard condition and a value is chosen in such a way that the condition evaluates to false and the loop is exited.

In comparison to the Levenshtein Distance when it comes to instructions, the Levenshtein distance for the data is not very large, especially considering the number of bytes that are compared (4096). This further indicates that while the number of executed instructions is lower, the state that the emulator finds is very similar to the state computed by real hardware. The few bytes that do differ always come from underconstrained values that are read from peripherals. Even though the relevant constraints on this value are met, the value itself is unrealistic. However, because of the constraints are met, the value is not too different as to cause unwanted behaviour.

7.1.3. Jetset

The runtime listed in Table 6.5 indicates that Jetset is much faster on the provided samples, roughly 20 and 10 times. While these measurements were taken on different hardware due to time limitations, it is expected that the results obtained from the system described in this thesis would only be roughly twice as fast as currently measured. However, this still leaves Jetset significantly faster. In the other metrics, Jetset and this work have roughly the same results.

Table 6.4 shows that the path taken by Jetset is accurate to the path taken on real hardware. Comparing with Table 6.2 shows that the path that is taken is almost identical to the path taken by the system described in this thesis. The differences that do exist are due to early termination of long loops, where the symbolic execution is able to quickly detect that the emulation is waiting for a semaphore to be set by a peripheral, and thus set this semaphore after only a few iterations. The only difference in data is because an underconstrained symbolic value is saved to the stack.

Interestingly, it can also be seen that the system completes faster than the state of the art for these samples, even when the number of peripheral reads increases. This is likely because of a combination of the locality of the different tactics, the use of both symbolic and concrete execution, and the high proportion of peripheral communication in these samples.

The difference in speed is likely due to the difference in implementation - Jetset is built into the emulator itself, whereas the system described in this thesis is implemented based on emulator hooks. There is an increased overhead when executing these hooks. While the speed is a downside of using this system, an advantage to this system of hooks is the ease of extending the system by adding more hooks, which gives more flexibility.

The flexibility offered by this hook-based system allows researchers to alter various features. For example, logging functions can be hooked to instead write data to a file. Another option is to hook a function that reads data from some source. The hook could then read data from another (possibly random) source for use in a fuzzing setup.

As such, while the performance of the system described in this document appears to be worse than other systems such as Jetset, the alternative approach presented in this work does have its merits, in the form of increased flexibility.

7.2. Discussion

In general, it appears that the problem of handling peripheral communication is quite a difficult problem. Although this system performs quite well at the beginning of samples, it generally gets stuck at some point. This seems to be because of underconstrained peripheral values that are stored in the concrete state. At some point, these values are loaded again and additional checks are performed. Although theoretically, the system should eventually find a value that passes even these checks, the amount of backtracking required makes this very impractical.

The timing information collected as part of the comparison with the state of the art was not usable to draw hard conclusions about the exact performance of the method described in this thesis compared to the state of the art. While still giving an indication, a more thorough timing analysis is desirable and might shine some light on other strengths or weaknesses of the different approaches.

7.3. Future Work

There are several directions for future work to further improve this system and advance the state of the art in this field of research.

7.3.1. Loop Detection

Detecting whether a system has reached an infinite loop is a notoriously difficult problem. In fact, it was even proven to be undecidable in the general case [48]. However, a combination of heuristics can be used to provide an approximate solution. Although the heuristic used in this thesis turned out to work reasonably well, it still fails in some scenarios that are not infrequently encountered in the real world. Especially in cases where a loop contains a read from a peripheral, and the loop guard depends on a concrete value, the heuristic that was used falsely indicates an infinite loop.

An alternative approach might employ symbolic execution to calculate the guard of such a loop, but the added cost (in time and computational power) needs to be balanced against the effectivity. Employing too much symbolic execution will likely drastically reduce the speed (and thus the utility) of a system.

7.3.2. Delayed Concretisation

Another pitfall of the system described in this thesis concerns underconstrained peripheral values that are copied into concrete memory. Since these values are underconstrained, the system chooses an arbitrary value that satisfies all the constraints. However, after this value is copied to concrete memory, at a significantly later point in the execution, further checks might be performed on this value. It is likely that these checks fail. If there are many other, unrelated, peripheral reads, the backtracking system will take a prohibitively long time to backtrack to the peripheral read that caused the wrong value. Even then, it is unlikely that the underconstrained value will be resolved to a value that does pass the additional, delayed checks.

As such, an approach that delays the concretisation of peripheral reads in such cases, effectively storing symbolic values in concrete memory, could perform well in this regard. However, the increased use and mixing of symbolic and concrete values might not only provide technical difficulties, but also lead to an undesirably high extra runtime.

7.3.3. Tighter Integration

It appears that there is significant overhead caused by the hook-based system that is employed in the implementation of this system. However, this hook-based nature is not integral to the method presented in this thesis. As such, it might be possible to devise a hybrid system, that can be easily extended with hooks, while simultaneously having tightly integrated support for handling unknown peripheral communication.

7.3.4. Parallel Evaluation of Tactics

Currently, the method of evaluating which tactic is used at a certain point is quite simple. Future work might find that making this selection process more complex is beneficial. One interesting direction in particular is the parallel evaluation of different tactics and how well each would perform in a given instance.

7.3.5. Reduced Communication with Analysis Tool

In the method described in this thesis, communication with the analysis tool occurs at least once for every read from an address that is part of a peripheral. Since this analysis tool is a different process, this interprocess communication is a source of overhead. Future work might explore more efficient ways to communicate with the analysis tool, to reduce the number of times communication is necessary.

References

[1] Haniel Barbosa et al. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9.

[2] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, Aug. 2008, pp. 209–224.

[4] Chen Cao et al. “Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation”. In: *Annual Computer Security Applications Conference*. ACSAC ’20. Austin, USA: Association for Computing Machinery, 2020, pp. 746–759. ISBN: 9781450388580. DOI: 10.1145/3427228.3427280. URL: https://doi.org/10.1145/3427228.3427280.

[5] Daming D. Chen et al. “Towards Automated Dynamic Analysis for Linux-based Embedded Firmware”. In: *NDSS*. Vol. 1. 2016, pp. 1–1. DOI: 10.14722/ndss.2016.23415.

[6] Abraham A. Clements et al. “HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1201–1218. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[7] Abraham A. Clements et al. “Is Your Firmware Real or Re-Hosted? A case study in re-hosting VxWorks control system firmware”. In: 2021.

[8] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[9] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software”. In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC’18. Baltimore, MD, USA: USENIX Association, 2018, pp. 309–326. ISBN: 9781931971461.

[10] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. “Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. Xi’an, China: Association for Computing Machinery, 2016, pp. 437–448. ISBN: 9781450342339. DOI: 10.1145/2897845.2897900. URL: https://doi.org/10.1145/2897845.2897900.

[11] Drew Davidson et al. “FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution”. In: *USENIX Security Symposium*. Aug. 2013, pp. 463–478.

[12] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[13] Adam Fabio. *Killed By A Machine: The Therac-25*. Oct. 2015. URL: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/ (visited on June 14, 2022).

[14] Andrew Fasano et al. "SoK: Enabling Security Analyses of Embedded Systems via Rehosting". In: May 2021, pp. 687–701. DOI: 10.1145/3433210.3453093. URL: https://doi.org/10.1145/3433210.3453093.

[15] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[16] Grove Karl Gilbert. *Finley's tornado predictions*. Burr, July 1884. URL: https://www.worldcat.org/title/finleys-tornado-predictions/oclc/35804644.

[17] Patrice Godefroid. "Compositional Dynamic Test Generation". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: Association for Computing Machinery, 2007, pp. 47–54. ISBN: 1595935754. DOI: 10.1145/1190216.1190226. URL: https://doi.org/10.1145/1190216.1190226.

[18] Fabio Gritti et al. "SYMBION: Interleaving Symbolic with Concrete Execution". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–10. DOI: 10.1109/CNS48642.2020.9162164.

[19] Zhijie Gui et al. "FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution". In: *IEEE Access* 8 (2020), pp. 29826–29841. DOI: 10.1109/ACCESS.2020.2973043.

[20] Eric Gustafson et al. "Toward the Analysis of Embedded Firmware through Automated Rehosting". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 135–150. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[21] Lee Harrison et al. "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 789–806. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[22] Roee Hay. "Fastboot Oem Vuln: Android Bootloader Vulnerabilities in Vendor Customizations". In: *Proceedings of the 11th USENIX Conference on Offensive Technologies*. WOOT'17. Vancouver, BC, Canada: USENIX Association, Aug. 2017, p. 22. DOI: 10.5555/3154768.3154790. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/hay.

[23] Grant Hernandez et al. "FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2245–2262. ISBN: 9781450349468. DOI: 10.1145/3133956.3134050. URL: https://doi.org/10.1145/3133956.3134050.

[24] Grant Hernandez et al. "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware". In: *Symposium on Network and Distributed System Security (NDSS)*. 2022.

[25] IONOS. *What is a bootloader and how does it work?* May 2022. URL: https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/ (visited on May 24, 2022).

[26] Paul Jaccard. "THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1". In: *New Phytologist* 11.2 (1912), pp. 37–50. DOI: https://doi.org/10.1111/j.1469-8137.1912.tb05611.x. eprint: https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x. URL: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x.

[27] Evan Johnson et al. "Jetset: Targeted Firmware Rehosting for Embedded Systems". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 321–338. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson.

[28] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. “Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation”. In: *SECURWARE 2016*. 2016.

[29] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. “Prospect: Peripheral Proxying Supported Embedded Code Testing”. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 329–340. ISBN: 9781450328005. DOI: 10.1145/2590296.2590301. URL: https://doi.org/10.1145/2590296.2590301.

[30] Mingeun Kim et al. “FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis”. In: *Annual Computer Security Applications Conference*. Association for Computing Machinery, Jan. 2020, pp. 733–745. ISBN: 9781450388580. DOI: 10.1145/3427228.3427294. URL: https://doi.org/10.1145/3427228.3427294.

[31] Karl Koscher, Tadayoshi Kohno, and David Molnar. “SURROGATES: Enabling near-Real-Time Dynamic Analyses of Embedded Systems”. In: *Proceedings of the 9th USENIX Conference on Offensive Technologies*. WOOT’15. Washington, D.C.: USENIX Association, 2015, p. 7.

[32] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: Association for Computing Machinery, 2012, pp. 193–204. ISBN: 9781450312059. DOI: 10.1145/2254064.2254088. URL: https://doi.org/10.1145/2254064.2254088.

[33] Hao Li et al. “FEMU: A firmware-based emulation framework for SoC verification”. In: *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2010, pp. 257–266.

[34] Wenqiang Li et al. “From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware”. In: Jan. 2021. DOI: 10.14722/ndss.2021.24308.

[35] Jacques-Louis Lions et al. “Ariane 501 Inquiry Board report”. In: July 1996. URL: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[36] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. “Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387556. URL: https://doi.org/10.1145/3342195.3387556.

[37] Jamie Lynch. *The Worst Computer Bugs in History: The Ariane 5 Disaster*. Sept. 2017. URL: https://www.bugsnag.com/blog/bug-day-ariane-5-disaster (visited on June 14, 2022).

[38] Dominik Maier, Lukas Seidel, and Shinjo Park. “BaseSAFE: baseband sanitized fuzzing through emulation”. In: July 2020, pp. 122–132. DOI: 10.1145/3395351.3399360.

[39] Alejandro Mera et al. “DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis”. In: (2020). DOI: 10.48550/ARXIV.2007.01502. URL: https://arxiv.org/abs/2007.01502.

[40] Marius Muench et al. “Avatar²: A multi-target orchestration platform”. In: *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*. Ed. by ISOC. San Diego, 2018.

[41] Anh Quynh Nguyen and Hoang Vu Dang. *Unicorn: Next Generation CPU Emulator Framework*. Aug. 2015. URL: https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf (visited on June 14, 2022).

[42] Ralf Nico. *LuaQEMU*. 2017. URL: https://github.com/comsecuris/luaqemu (visited on Feb. 24, 2022).

[43] Jan Ruge et al. “Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge.

[44] Yan Shoshitaishvili et al. “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware”. In: Jan. 2015. DOI: 10.14722/ndss.2015.23294. URL: https://doi.org/10.14722/ndss.2015.23294.

[45] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. “The value flow graph: A program representation for optimal program transformations”. In: *ESOP ’90*. Ed. by Neil Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 389–405. ISBN: 978-3-540-47045-8.

[46] Seyed Mohammadjavad Seyed Talebi et al. “Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 291–307. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/talebi.

[47] T. T. Tanimoto. *An elementary mathematical theory of classification and prediction by T.T. Tanimoto*. English. International Business Machines Corporation New York, Nov. 1958, 10 p.

[48] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[49] Jonas Zaddach et al. “Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems’ Firmwares”. In: Feb. 2014. ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23229.

[50] Yaowen Zheng et al. “FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

[51] В. И. Левенштейн. “Двоичные коды с исправлением выпадений, вставок и замещений символов”. In: *Докл. АН СССР* 163 (4 1965), pp. 845–848. URL: http://mi.mathnet.ru/dan31411.

A
Bugs in Open Source Projects

While implementing the system described in this thesis, several bugs in different open source projects were found. In some cases, fixes were submitted to the repository.

A.1. Qiling

Qiling is a cross-platform emulator that supports many different architectures. It builds on top of Unicorn [41], a fork of the QEMU emulator [2]. However, it tries to set itself apart by intending to enable developers to use it as a framework to build their own systems on top of it. To this end, it provides several utilities to handle things like memory, operating systems and more.

We found several bugs in their implementation of memory utility functions, and missing register names for several architectures. Although we did not submit Pull Requests to fix these issues, we did outline a proposed solution in all of the created issues.

1. **`ql.mem.search` search uses regular expression**

   `ql.mem.search` raises an exception for some input needles, and wrong results for other needles. This is because the needle is interpreted as a regular expression, which is not documented anywhere and quite unusual.

   url: `https://github.com/qilingframework/qiling/issues/1135`

2. **Restoring a state with MMIO causes an exception**

   Mapping an MMIO region, then saving the qiling state and then restoring the qiling state causes an exception.

   url: `https://github.com/qilingframework/qiling/issues/1136`

3. **Unmapping part of a MMIO region causes future `ql.save`s to crash**

   When saving a qiling state after having unmapped part of an MMIO region, the save fails with a `KeyError`.

   url: `https://github.com/qilingframework/qiling/issues/1137`

4. **ARM64: `q?` registers are missing from the available register names**

   The ARM64 `q?` registers are missing from the available register names, so you cannot simply use `ql.reg.read("q0")` to read the `q0` register. In contrast, you can use `ql.reg.read("x0")` to read the `x0` register.

   url: `https://github.com/qilingframework/qiling/issues/1138`

5. **`ql.mem.unmap_all` does not unmap all memory regions**

   Mapping a memory region starting at address 0 and then calling `ql.mem.unmap_all` does not unmap the region.

   url: `https://github.com/qilingframework/qiling/issues/1141`

6. **ARM: `d?` registers are missing from the available register names**

   The ARM `d?` registers are missing from the available register names, so you cannot simply use `ql.reg.read("d0")` to read the `d0` register. In contrast, you can use `ql.reg.read("r0")` to read the `r0` register.

   url: `https://github.com/qilingframework/qiling/issues/1179`

A.2. Claripy

Claripy is an abstraction layer for constraint solvers, specifically the Z3 solver by De Moura and Bjørner [12]. Claripy simplifies the process of working with the Z3 solver by providing wrappers for common functions, allowing users to use regular Python syntax to manipulate symbolic variables and add constraints.

Claripy is used in the system described in this thesis when transforming the concrete state to a symbolic state, and while working on this, we found two inefficiencies that caused the state transfers to be significantly slower than needed. We submitted two pull requests, which were promptly merged.

1. **`claripy.Extract` uses exponentation instead of a binary shift**

   While profiling some code that used some rather large concrete bitvectors, we found that a lot of time was spent in `claripy.Extract`. We found that the main culprit is the use of exponentiation to construct a bitmask instead of a shift. We show that changing this exponentiation by a shift can result in a big increase in performance when the parameter values to `claripy.Extract` are large.

   url: `https://github.com/angr/claripy/issues/267`
   PR: `https://github.com/angr/claripy/pull/268`

2. **`claripy.Extract` uses a bitmask with too many bits set**

   `claripy.Extract` currently uses a mask of `f` 1-bits, and the resulting value is put in a bitvector with only `f - t + 1` bits. The result is the same if only `f - t + 1` bits were used in the initial mask. This PR makes the mask just as big as it should be to avoid another truncation when the bitvector is constructed. This improves the performance if `t` is large.

   PR: `https://github.com/angr/claripy/pull/269`