Trade-offs of the SotA automatic vulnerability detection techniques on ARM

van den Elzen, S.W.

Award date:
2019

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Trade-offs of the SotA automatic vulnerability detection techniques on ARM

Master Thesis

Sjors van den Elzen

Supervisors:
dr. Luca Allodi
Robin Vossen, Bsc.
Razvan Venter, MSc.

Final version

Eindhoven, October 2019

Abstract

Embedded devices are present in most households, offices, factories, and critical infrastructure. With the surge of interest in the Internet of Things, the number of devices in use will likely continue increasing for years to come. Since any of these devices is essentially a small computer, all of them can pose security risks as well. Finding these security risks in embedded devices depends on analysing machine code, as the software is generally proprietary. Analysing the behaviour and security of software is a well established field, whereas automatic analysis without the source code or design of the software is relatively new. We look at several existing analysis techniques, and compare their effectiveness in the locating of such security risks in embedded devices. As there are many different types of embedded devices, in order to make a significant difference in the security of these devices, these techniques have to be able to also scale to those many different devices. Therefore these techniques could not rely on having access to the actual device, as that would not scale well, and they need to be fully automated. We have looked at three such techniques; symbolic execution, taint analysis, and fuzzing. We qualitatively analysed the results from state of the art frameworks that implement one or more of these techniques, and are usable on binary code. While some techniques were able to confirm vulnerabilities in our ground truth, said ground truth was too small to draw any concrete conclusion.

Preface

First I would like to thank my supervisors who supported me in writing this thesis. Luca Allodi for sitting down with me many times to discuss the methodology and overall approach to the thesis. Robin Vossen for sharing his knowledge of binary analysis and reverse engineering, which helped gain an understanding on them so much quicker. Razvan Venter for his advice on setting a scope for the thesis.

Thank you to my girlfriend Kelly Morales, who was there to support me throughout. Thank you to friends whom I could discuss all kinds of small technical problems with, and thank you to my family who listened while I was trying to explain to them what I do.

Lastly I would like to thank all the great people at Secura, with whom I could discuss all sorts of topics related to this thesis, and who are always ready to share their knowledge.

Contents

1 Introduction **1**
1.1 Problem statement . . . 1
1.2 Research goal & Question . . . 1
1.3 Scope . . . 1
1.4 Outline . . . 2

2 Preliminaries **3**
2.1 Disassembling and lifting . . . 3
2.2 Information gathering . . . 5
2.3 Finding vulnerabilities . . . 7

3 Background **11**
3.1 Related work . . . 11
3.2 Performance metrics . . . 12
3.3 Existing tools . . . 13

4 Methodology **17**
4.1 Firmware . . . 17
4.2 Analysis tool selection . . . 19
4.3 Confirming found vulnerabilities . . . 19
4.4 Experiment setup . . . 20
4.5 Methodological limitations . . . 20

5 Results **23**
5.1 Analysis results . . . 23
5.2 Qualitative analysis . . . 23
5.3 Discussion . . . 26

6 Conclusions **29**
6.1 Future work . . . 29

Appendix **35**

A CVE locations **35**

List of Figures

1 Example flow of how an analyst uses the described binary analysis techniques. . . 4

2 The setup of the experiment. 21

List of Tables

1 Vulnerabilities in the Internet of Things (IoT) related to the firmware itself. 13

2 Categories of firmware targets, i.e. the types devices in the IoT 17
3 Features of the state of the art binary analysis tools 19
4 The ground truth resulting from our criteria in section 4.1 21

5 Summarised results of the analyses. 24
6 Number of matches in the results from BAP's analyses. 25

7 The exact addresses and ranges for each Common Vulnerability and Exposure (CVE) in each product. 35

Listings

1 Example code, demonstrating a state explosion. 9
2 Code snippet from the license upgrader, showing the buffer overflow. 25
3 Code snippet from the crypto library, showing the out-of-bounds read. 26
4 Code snippet from the disk utility library, showing the buffer overflow. 26

List of Acronyms

ABI Applied Binary Interface

AFL American Fuzzy Lop

ARM Advanced RISC Machine

BANG Binary Analysis Next Generation

BAP Binary Analysis Platform

BARF Binary Analysis and Reverse engineering Framework

BIL BAP IL

CFG Control Flow Graph

CFI Control Flow Integrity

CPU Central Processor Unit

CTI Control Transfer Instruction

CVE Common Vulnerability and Exposure

CWE Common Weakness Enumeration

DTA Dynamic Taint Analysis

FTP File Transfer Protocol

GNU GNU's Not Unix

IL Intermediate Language

IoT Internet of Things

ISA Instruction Set Architecture

MIPS Microprocessor without Interlocked Pipelined Stages

NVD National Vulnerability Database

OS Operating System

PLC Programmable Logic Controller

QEMU Quick Emulator

REIL Reverse Engineering Intermediate Language

ROP Return Oriented programming

RTOS Real Time Operating System

SAT Boolean Satisfiability

SMT Satisfiability Modulo Theories

SOHO Small Office / Home Office

S^{2}E Selective Symbolic Execution

STA Static Taint Analysis

VFG Value Flow Graph

VM Virtual Machine

VSA Value Set Analysis

Chapter 1

Introduction

1.1 Problem statement

Analysing the binaries of software is as old as binaries themselves, and is used to debug, to find vulnerabilities, to learn what an unknown program does, or to update legacy software when the source code is no longer available. While the problem of analysing binaries has been thoroughly explored on x86 / x86-64 (x64 for short) architectures [1], on other architectures it is less well researched. Even as the IoT is becoming more relevant by the year, software on processors that are typically used for embedded implementations is still not as thoroughly tested by binary analysis frameworks. Since more people start using 'smart' devices, more opportunities arise for malicious hackers. In order to counteract this, new tools have to be created to assist security researchers in finding vulnerabilities in embedded software. An example of such a tool is the Capstone disassembly framework [2], which is an open source disassembly framework based on the LLVM (not an acronym) project. Because of this dependency it is able to support a wide variety of architectures, including some that are used in many embedded devices, such as the Advanced RISC Machine (ARM) and Microprocessor without Interlocked Pipelined Stages (MIPS) architectures. This enables other people to start building tools on top of it, but this development still lags behind the research on x86 / x64.

1.2 Research goal & Question

In order to improve on the weak points of binary analysis on IoT firmware, these weak points first need to be identified. This thesis aims to give a qualitative analysis of the current state of the art, laying existing analysis techniques side by side in order to compare their effectiveness in finding vulnerabilities. This goal can be summarised in our research question:

> **Research question:** What are the advantages and disadvantages of state of the art automatic vulnerability detection techniques on ARM?

1.3 Scope

Since the amount of architectures, devices, and software that can be classified as being part of the IoT is enormous, this section aims to limit the scope to a size that allows some meaningful analysis. The important choices to make are picking an architecture, an instruction set compatible with this architecture, whether the target firmware is based on an operating system, and what format the firmware comes in when handing it over to the different tools.

1.3.1 Input

Analysing the software of real-world devices comes with several challenges. Firstly, firmware is usually *packaged*, which means one or multiple binaries are put in one file, together with a file system, and possibly with metadata, which can be in any format the manufacturer uses. Since different manufacturers often use different ways of packaging firmware, unpacking firmware is a non-trivial problem. Second, assuming the firmware has been unpacked, there are one or more

binaries to analyse. These usually don't contain information regarding their functionality other than the binary code itself (i.e. symbolic information), leaving another obstacle. Lastly, the disassembly of these binaries can be difficult and sometimes inaccurate (also see section 2.1). After disassembling the code, assembly code is left, its syntax depending on the architecture it was originally written or compiled for. While it is possible to directly analyse this code, or even a binary itself, most frameworks opt to further process the program before doing so. This is a step of abstraction, leading to an Intermediate Language (IL), which is usually a low level and easy to analyse language. While assuming the input consists of programs in such a language would certainly make our work easier, it is not a realistic assumption. Therefore the scope includes 'raw' binary code, which has been unpacked from a firmware image by external tools. See section 4.4 for more details.

1.3.2 Instruction Set Architecture

Many IoT devices are based on the ARM architecture [3], especially ones that have a relatively wide range of functionality, such as smart TVs, routers, or thermostats. As such, the architecture is a good choice for the firmware we base the verification of the research on. ARM itself defines different Instruction Set Architecture (ISA)s, and at the time of writing the actively used ones are the Thumb-2, native ARM, and the Aarch64 instruction sets. These are 16-bit, a 32-bit, and a 64-bit instruction sets, respectively. Despite their size difference, Thumb-2 and Native ARM are similar, to the point where an optimised program will interleave them. These two are the ISAs used in most firmware, and will be in scope. Aarch64 is a very different ISA, and as such will be left out of scope.

1.3.3 Embedded level

The IoT (mostly) consists of embedded devices, which are devices with a dedicated functionality, such as measuring the temperature. In this aspect the devices still vary greatly however, as some are 'more embedded' than others. One way to distinguish them is by looking at how close to the physical device the firmware operates. *Deeply embedded devices* [4] operate on bare metal or an extremely minimal Operating System (OS), allowing them to very efficiently perform one specific task. Firmware based on some larger OS, like Linux, is usually more general purpose. Such an OS can also be a Real Time Operating System (RTOS). This can enforce deadlines on tasks, giving guarantees on the timeliness of the system. This can change the way an exploit works on such systems, making it time sensitive. Therefore the detection of vulnerabilities can also vary, which makes it unsuitable for our research goal. By choosing deeply embedded firmware, building a representative data set would become very difficult, as very little assumptions can be made about the packed firmware when unpacking it, making automation of this process very difficult. These systems are also more often dependent on peripherals, or are peripherals themselves, complicating a proper analysis. For these reasons the firmware in our data set should be based on a (non Real Time) OS.

1.4 Outline

The second chapter will detail the terms used and techniques studied in this thesis. In the third chapter, existing literature on the topic is reviewed. This includes similar works in literature, types of vulnerabilities found in binary code, and existing binary analysis implementations. The fourth chapters sets out a methodology to answer the research question. The fifth chapter presents the results acquired with this methodology, and discusses them. In the sixth and final chapter we draw a conclusion based on the fifth chapter, answering our research question.

Chapter 2

Preliminaries

This section defines several concepts that are at the core of this thesis. While most are generally accepted definitions, we reiterate them here to ensure a common definition with the reader.

Definition 1. Branch: A branch is an execution path pointed to by a conditional statement in a software's code. Thus for every conditional statement there are two branches.

Definition 2. Bug: A bug is an error in the code that was not put there intentionally, and can put the program in an unintended state.

Definition 3. Vulnerability: A vulnerability is a bug in the code that could be abused by an attacker to violate a system or software security policy.

Definition 4. Exploit: An exploit is some code or action that makes use of a vulnerability in a program to achieve the attackers goal.

Definition 5. Firmware: Firmware is the combination of computer instructions and data that reside as read-only software on an embedded device (different from, but based on [5]).

Automatically finding vulnerabilities in binary code is dependent on several techniques that allow an analyst to gain information about the code under investigation. These techniques can be roughly categorised in translating the machine code to a more understandable format, in section 2.1, finding out how the code functions, in section 2.2, and finding vulnerabilities in the code, in section 2.3. A work flow with these techniques can be found in figure 1.

2.1 Disassembling and lifting

The problem of disassembling binary code is not a trivial one. In source code information such as function and data structure names is available, called the symbolic information. Usually all symbolic information will be stripped from the code in the assembling process, and as such it is left to the analyst to find out what each variable or data structure is and represents. As each instruction effectively consists of a series of ones and zeroes, binary code can overlap in memory as well, if the end of one such series shares space with the beginning of the next one. According to Andriesse [1], however, on x86/x64 this does not happen outside highly optimised libraries. To the best of our knowledge, this does not happen in ARM architectures at all, even in optimised code. Aside from these points, disassembly is different for each architecture, as the ISA is different for all. The ISA is the definition of how machine code maps to instructions for the Central Processor Unit (CPU), where each series of machine code mapped to an instruction is called an *opcode*, and each human-readable instruction a *mnemonic*. Disassembly being different on each ISA means an approach to disassembly that works well in one architecture does not necessarily work well on another architecture. Lastly, if a disassembly of the binary code is acquired, it is non-trivial to tell whether it is the correct disassembly. Especially on dense instruction sets, where most possible opcodes encode some instruction, Andriesse suggests it is difficult to (automatically) detect mistakes. Part of this difficulty also comes from indirect control flows, which are explained in section 2.2.1.

In order to tackle these problems and take the first step in reverse engineering a binary, let us take a look at the different types of disassembly. All disassembly can be split up in two different types:

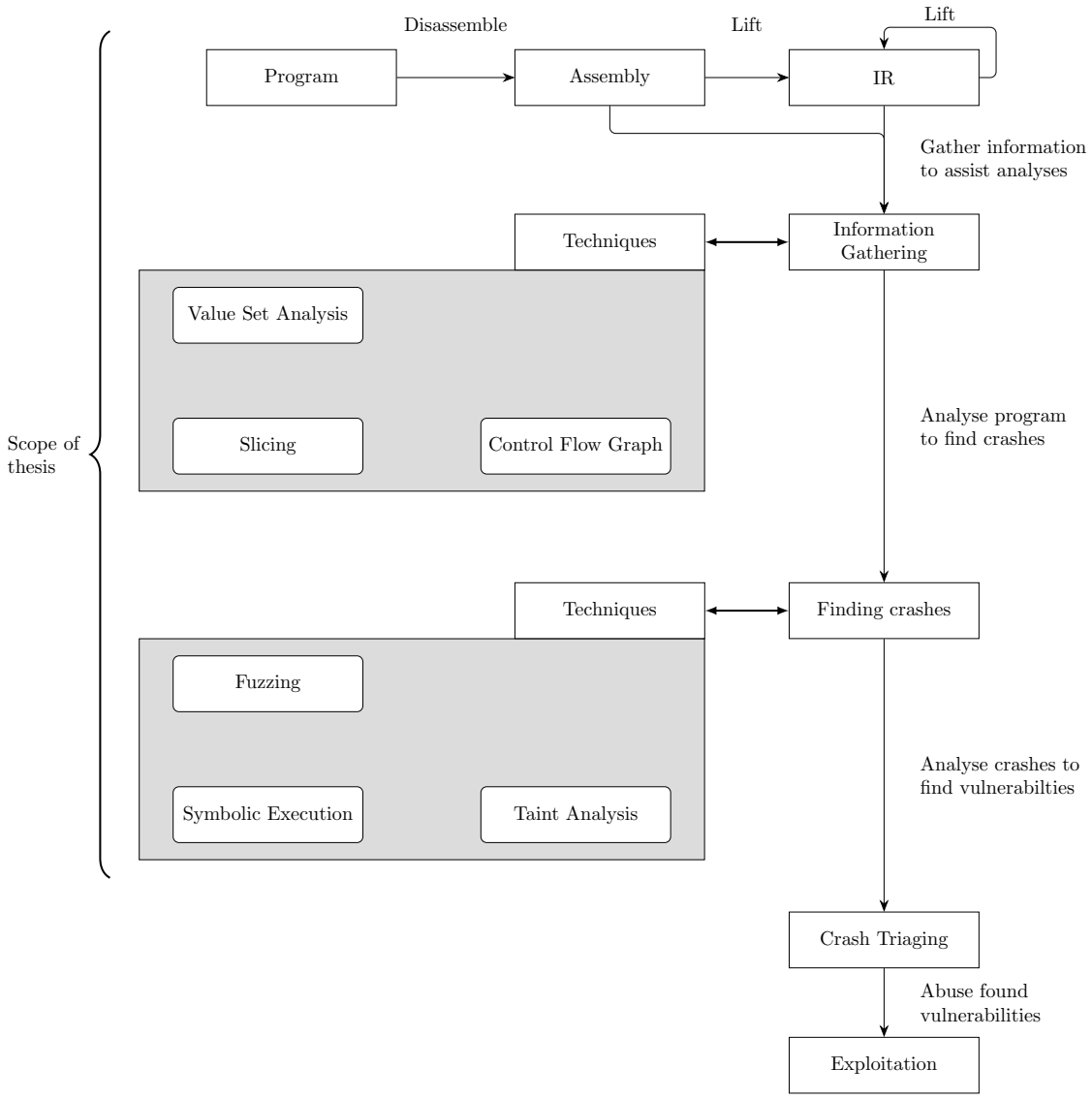

Figure 1: Example flow of how an analyst uses the described binary analysis techniques.

static disassembly and *dynamic disassembly*. Static disassembly processes the binary without running it, and dynamic disassembly involves running the binary in some way. Running the binary has the clear advantage of being able to simply trace the execution path, and thus does not have to reason about the control flow of the program. Running the code is not always an option however. For example in the case of deeply embedded firmware, it is usually not possible as the firmware is very dependent on the environment of the device it would normally run on. Since binary analysis can be very computationally intensive, it is usually done on processors with a lot of power. High power processors have a different architecture from embedded processors, so embedded firmware can't run on it natively. This can be resolved by running the code in an emulator, but even a good emulator is only an approximation. Static disassembly can be split further into two different types, namely *linear* and *recursive* disassembly. In a linear mode the disassembler starts at the beginning of the binary, translating the first valid opcode it can find, and then moving to the next one until it reaches the end of the binary. This will never detect overlapping instructions, as the start of the next instruction is selected based on the ending of the current one. Recursive disassembly also starts at the beginning, but whenever a conditional statement is detected, it follows both branches where possible. This means overlapping instructions will be found if they are referred to at any point in the code, but any code not referenced by already discovered code will remain undetected. As opposed to a dynamic disassembler, a recursive static disassembler has to reason about variables in the code, since these do not get computed on an actual or emulated CPU. If the disassembler is not able to resolve the address of a branch that will also result in that branch not being disassembled, unless it is directly referred to at some other point in the code.

2.2 Information gathering

Information gathering techniques can be used to derive insight in the functionality of code. This can be limited to finding out what the code does, or extended to finding out what the code does *wrong*. This implies some design behind the code should be found, detailing how the code should work. At the same time deviations from this design should be found, from the same piece of code. Assuming a correct or accurate enough representation in Assembly is found using the techniques described in the previous section, any next step depends on the goal of the analyst. Generally when trying to find exploitable behaviour, some kind of IL is made before analysing the code further. This is called 'lifting' the code, and is done since it reduces the complexity of further analysis. Most ILs only have instructions that are completely atomic, (e.g. an addition does not set flags itself), thus reducing the complexity per instruction. In order to gain a first understanding of any program, most people will first look at how different parts are put together. This is the case for binary analysis as well. For both manual and automatic analyses it is common to start with finding the control flow of the program, which is further detailed in section 2.2.1. For some dynamic approaches to problems it can be useful to simply run the program and see what it does. In order to see what a program does, it needs to be *instrumented*. Binary instrumentation is explained in section 2.2.2. Sometimes an analysis is infeasible when a program is analysed as whole, but not when a certain part is taken out. Section 2.2.3 describes how this can be achieved. Finally, a method to estimate bounds on indirect control flows in described in section 2.2.4

2.2.1 Control flow

When trying to understand what a program does, a Control Flow Graph (CFG) helps tremendously. The control flow is the entire path an execution takes through the code. Most research uses *basic blocks* as a term for a number of instructions that follow each other, and do not alter this control flow. A CFG can be built from all basic blocks in the code, where each node in the graph is a basic block. More formally we have definitions, slightly altered, from [6]:

Definition 6. Control Transfer Instruction: A Control Transfer Instruction (CTI) is an instruction that directs the control flow of the program.

Definition 7. Basic Block boundary: A basic block boundary is a point in-between two instructions where either of these instructions is a CTI or a target of a CTI.

Definition 8. Basic Block: A basic block is a sequence of consecutive instructions preceded and succeeded by a basic block boundary, with no basic block boundary in-between.

Definition 9. Control Flow Graph: Given a program P, its control flow graph is a directed graph $G = (V, E)$, where V is the set of basic blocks and $E \subseteq V \times V$ is the set of edges representing control flow between basic blocks. A control flow edge from blocks u to v, $u, v \in V$, is $e = (u, v) \in E$.

Retrieving a CFG is extremely useful. Since most binaries are stripped of their symbolic information, it is not immediately clear where functions start and end. By building a CFG, this information can be retrieved and then be used to get a first impression of the workings of the binary. The approaches to building it vary slightly, as again there are static and dynamic methods, but most are a bit of both. Similarly to disassembly, a static analysis can be linear or recursive. Since the approach is the same as for disassembly, we will not discuss this again here. With a dynamic approach every CTI is instrumented (see the next section), such that every basic block can be distinguished. When running the program with varying inputs, most execution paths should be uncovered and a CFG can then be constructed by observing each CTI. A good example of a static CFG recovery is the Binary Analysis and Reverse engineering Framework (BARF) [7], which allows both a linear and a recursive method. Alternatively a dynamic method similar to symbolic execution is possible, Forced Execution [6]. With this method, at every branch both possibilities are executed, resulting in a more complete CFG.

2.2.2 Binary instrumentation

When analysing a program dynamically, the behaviour needs to be recorded somehow. This can be done dynamically by running the entire program in an emulator where the fake processor keeps track of all instructions it executes, or it can be done on a real processor by adding instructions to the program that communicate information about the state of the program. Binary instrumentation encapsulates instructions or functions that are of interest to the analyst with code that gives a clue about the semantic meaning of what is happening. This can be very simple, by for example only logging the beginning and ending of a function. It can also be more complicated, such as reading out parts of memory when certain instructions are run. Additionally, if all CTI (see definition 6) are instrumented, it is possible to map the control flow of a program. Since this avoids interrupting the execution with checks, such as in a debugger, it allows for relatively quick analyses.

Examples of state of the art binary instrumentation frameworks are Intel Pintool [8], DynamoRIO [9], and Valgrind [10]. These are all used by binary analysis frameworks, such as angr 3.3.1 and Triton 3.3.4.

2.2.3 Slicing

Slicing is a technique that 'cuts out' a path or set of paths from a program, based on a property of those paths. It is generally used to find all instructions that influence a certain variable, by analysing the program from an exit point back to the entry point and putting every instruction that affects the specified variable in the slice. This type of slicing is called backward slicing, for the order of traversing the program. Similarly, forward slicing aims to find all instructions that are influenced by a given variable by walking from an entry point to an exit point. Slicing is used to limit the number of instructions to be analysed, which is useful when performing computationally intensive techniques such as symbolic execution (see section 2.3.3).

2.2.4 Value Set Analysis

Value Set Analysis (VSA) [11] is a technique used in static analyses used to approximate all values pointers can have in order to resolve indirect control flows. Since a static analysis has to reason about control flow itself, as opposed to dynamic analyses where the processor does this, it is necessary to create techniques that assist this reasoning. VSA achieves this using *abstract locations*, which are the set of all memory locations a pointer can point to. An over-approximation of an abstract location is computed instead of resolving all possible values, as this is done at every computation step, and thus the exact possibilities can not be found in a reasonable amount of time. In the end, though still useful, VSA therefore only gives an educated guess at what a pointer can point to. See section 3.3.1 for an application of VSA.

2.3 Finding vulnerabilities

When automating the search for vulnerabilities, there are several techniques that can be used. While all previously discussed techniques aid the analyst in understanding the semantics of a program, none of them actually finds vulnerabilities. In this section we review techniques that do. The choice for a technique depends on multiple factors: the type of vulnerability to look for, the size of the program, and the platform it runs on. While all techniques can be applied in most situations, some are more useful than others when faced with certain constraints. For example, if a program has many important variables to keep track of, symbolic execution (on its own) will not work as well as with smaller programs, since it will run out of memory quickly. It is much more useful when only a very specific input will trigger the vulnerability, a case where fuzzing would yield results very slowly.

2.3.1 Fuzzing

Fuzzing is a technique first introduced in an analysis of UNIX utilities [12]. It is based on the generation of random data in inputs to a program, in an effort to crash the program. Since this first concept fuzzing has evolved, introducing new types of fuzzing such as whitebox fuzzing [13]. Whereas originally the fuzzing technique approaches the system as a 'black box', whitebox fuzzing uses knowledge of the target, such as source code, to fuzz the target more effectively. Godefroid et al. accomplish this by combining fuzzing with symbolic execution (see section 2.3.3). By finding constraints on inputs that allow the program to reach certain states, the fuzzing can generate inputs that always allow execution to continue until deep into the program. Thus it can find vulnerabilities lying deep in the program more quickly than a fuzzer which has to rely on generating an input that coincidentally drives execution that far. Another variant of fuzzing is mutational fuzzing, which starts with valid inputs to the program and slightly alters these inputs until something interesting happens, like a crash. One such fuzzer is American Fuzzy Lop (AFL) [14], which instruments the source code and then compiles it, thus having access to all information it needs when running and fuzzing the program. It will be further discussed in section 3.3.5.

While whitebox fuzzing is very effective, it does require source code to achieve its results. Since our target is binary code, this will not work. After using some of the analyses that gather information, however, enough information is available to achieve more than simple black box fuzzing would. AFL supports fuzzing binary code by emulating the binary with Quick Emulator (QEMU), and accessing the information on the fake processor instead of instrumenting the program code. This is limited in that it relies on the QEMU user mode, which tries to naively run the binary with no environment context, and only fuzzes the binary itself. Any shared libraries, some of which are proprietary in the case of firmware, are ignored. There also exists an (unsupported) extension on AFL called AFL-Unicorn [15], using the Unicorn emulator [16]. Unicorn is based on QEMU system emulation, but strips away everything not directly related to the CPU. In terms of accuracy it thus ends somewhere between QEMU user mode and system mode. AFL-Unicorn uses a memory dump from a debugger to approximate the state the binary should start in, generates an input

which is stored in said memory, and then emulates it in Unicorn until it crashes or times out. Since Unicorn only emulates the CPU, this does not need detailed information on the environment of the binary, such as peripherals. While this could mean the results are less accurate, it does mean all binary code can be tested.

An obvious downside to fuzzing is that the progress an analysis makes does not have to be clear. After generating an enormous amount of random inputs to a program, an analysis can have made no progress at all. This is unlikely though however, especially with mutational fuzzing. The simplicity of fuzzing is its largest advantage. Since generating random input and running a program is quite simple, this can be done many times before other techniques find something.

2.3.2 Taint analysis

When trying to find out whether input controlled by an attacker can reach exploitable behaviour in a program, the easiest way is to track this by adding metadata to all inputs and their derived variables. This technique is called *Taint Analysis*, and like most binary analysis techniques can be done statically or dynamically. Static Taint Analysis (STA) analyses the code without running it, and thus the tool used for the analysis should keep track of all metadata of inputs and their derivatives. In Dynamic Taint Analysis (DTA) the code is executed, so part of memory is required to store the metadata, and some kind of instrumentation (see section 2.2.2) to know when to update it. Taint analysis can be split into three separate tasks: defining taint sources, propagating taint, and defining taint sinks [17]. Taint sources should be picked based on the attacker model, for example by mistrusting all input provided by a user. Taint sinks are points in the program that are sensitive, such as indirect control flows. Taint propagation is determined by a set of rules, depending on how much information the analyst is trying to get from the analysis, and what kind. All taint can be tracked at a bit-level, requiring one bit per bit of data to indicate whether that data is tainted or not. Since this is usually seen as too much overhead in memory usage, a lower granularity such as tracking at byte-level is used (one bit per byte). A second choice is colouring the taint. It is often useful to track where *exactly* tainted data came from, which can be done by giving taint sources separate 'colours'. Every user input would then have its own colour, and every bit of metadata in an uncoloured system would be replaced by a number of bits equal to the number of colours to be used. Since one variable can be tainted by multiple colours, each colour requires one bit of metadata. Lastly, a taint propagation policy needs to be defined. For every instruction that operates on data there needs to be a rule on how the output is tainted by the input. A rule of thumb for taint propagation is that if a byte in the output is controlled by an input byte, the output byte takes the taint from the input byte.

Taint analysis takes a bit of a middle road in terms of computational intensiveness. It can be demanding on memory usage, especially if using a high accuracy with multiple colours. This can be limited by computing the union of all backward slices of taint sinks, because it can exclude some inputs and thus colours. The main advantage is that in as little as one run of a program it can be determined how it is possible to corrupt certain points lying deeper in the code, thus being very fast.

2.3.3 Symbolic execution

Symbolic Execution is a method of analysing program behaviour, where the state of the program at every point is kept track of with a symbolical representation. At every CTI, symbolic execution follows both branches, keeping track of the state of the program in both paths. Since a program usually has many CTIs, this can quickly become computationally intractable. For example in listing 1, tracking states naively will create two different states for every check of $i \ \% \ 10 == 0$, resulting in $2^{100} \approx 1.27 \times 10^{30}$ different states. So even though the program is very simple and will always yield the same result, the number of possible paths explodes, making it practically unattainable to explore every single path. The number of states can even increase to infinity for unbounded loops or recursive calls [18].

```
int a = 0;
for (int i = 0; i < 100; i++)
{
    if (i % 10 == 0)
        a++;
}
```

Listing 1: Example code, demonstrating a state explosion.

One way to tackle this problem is *concolic execution*, a portmanteau of {*conc*rete-symb*olic*} execution. In concolic execution, a concrete version as well as a symbolic version of the state is kept. This allows avoiding the state explosion, by choosing a branch to take based on the concrete state. While this risks not covering the entire program code, good results are still achievable with repeated runs on different input values [18]. Another approach to limiting the number of states is by pruning the tree of all possible states. Pruning can be done in several ways, including but not limited to a depth-first search, defining a 'summary' of all functions with effects to the symbolic state, or combining the symbolic execution with fuzzing. In a depth-first search a branch is analysed until it reaches an exit point before exploring the other branch. Summarising a function means the symbolic representations of all variables get changed based on what the function usually does, without actually executing the function. When combining symbolic execution with fuzzing, some paths can be cheaply fuzzed and determined to not be interesting, thus limiting the coverage of the analysis to paths that seem more prone to vulnerabilities. The goal of symbolic execution is to automatically detect the conditions required to follow each path in a program. In order to achieve this, at every conditional CTI the resulting states are constrained based on the condition of the CTI. When a set of constraints has been build, these constraints can be further analysed by handing them to a *constraint solver*. A constraint solver takes a number of variables and a set of constraints on those variables, and finds out whether it is possible to satisfy all given constraints. This is also called the Satisfiability Modulo Theories (SMT) problem for constraints on numbers, or Boolean Satisfiability (SAT) problem for Boolean constraints. If a certain set of constraints is satisfiable, a constraint solver can also provide a set of concrete values that satisfy those constraints. This can be used to generate tests with a wide coverage, to prove a certain statement in the code is reachable, or to help develop exploits.

RWset [19] is a way to perform the depth-first search to prune possible paths. RWset defines live variables, which are all variables used in the current path after the current instruction. It uses a complete depth-first traversal of all paths after the current instruction to find all such variables. Given the set of live variables at a certain point, it is possible to merge multiple executions that took different paths earlier in the program but will behave the same for the rest of the program. Thus the number of different paths becomes much lower. In [20] this method is expanded and then implemented in KLEE [21], a symbolic execution framework, and its performance is analysed. This results, according to the authors, in an average performance increase of 50.5×, with a median of 10×, making it very interesting when using symbolic execution. Instead of simply finding live variables, the authors find all lines of code that are relevant for the current execution, by constructing a graph of dependencies between instructions. By selecting instructions that have not been executed yet as relevant, and then pruning execution paths that do not execute relevant instructions or instructions that are depended on by relevant instructions, it achieves similar coverage and a greater speed than other symbolic execution techniques.

A more complete survey of symbolic execution techniques can be found in the survery of Baldoni et al [22]. The authors review several variants of symbolic execution, design choices for a symbolic execution engine, and problems that come with each design choice. By far the largest disadvantage of symbolic execution is that it is very computationally expensive and needs a lot of memory. However, when it is given enough resources to finish its job, the results are expected to be more complete than any of the other techniques.

Chapter 3

Background

This chapter covers the literature study of this thesis. First some related works are discussed, and their scope compared to the scope of our work. Secondly we look at performance metrics for binary analysis in the literature. Afterwards existing implementations of the state of the art are reviewed, exploring and comparing the relevant properties. Lastly an overview of the implementations that are within our scope is given.

3.1 Related work

This thesis compares the advantages and disadvantages of binary analysis techniques, differentiating between vulnerability types to see whether a technique is better suited for a certain type of vulnerability. Several other studies review existing binary analysis methods or the information gathering techniques they depend on. First some studies detailing challenges in binary analysis itself are discussed, as those challenges also present themselves in this thesis. Afterwards publications similar to our work are compared, focusing on the approach each has to its analysis.

Over the years different instrumentation frameworks have been published, the most relevant of which are compared in [23] (paper in Spanish). The metric for comparison is the slowdown the instrumentation introduces to the execution, measured for several programs. Between the frameworks compared, Valgrind [10], PIN [8], and DynamoRIO [9], the number of instructions in an instrumented program is four to twenty times larger than in the original program. These numbers are indicative to the running time of different binary analysis techniques, as symbolic execution and taint analysis are dependent on instrumenting the binary. Fuzzing is not necessarily dependant on instrumentation, although the more advanced modern variations do make use of it (see section 2.3.1).

Fuzzing binary code from embedded devices has its own challenges as well, as shown by Muench et al [24], who look at different classes of embedded devices and challenges that are unique to fuzzing firmware. They identify three separate classes of embedded devices, of which the first class coincides with our scope. For two widely used firmware images with known vulnerabilities, they then analyse the behaviour of every class of embedded device. The results are used to identify challenges in the fuzzing of embedded devices. Two out of three challenges are related to the performance of fuzzing, including overhead introduced by instrumentation. The third challenge is the most interesting, namely fault detection. Most fuzzing depends on the system crashing or displaying other behaviour that indicates something went wrong. On desktop systems this is not a problem due to the protection mechanisms in place, but this is usually not the case on embedded devices. If fuzzing the firmware in an emulator, this can be mitigated by detecting conditions that would normally cause a crash through said emulator.

One large scale analysis of embedded firmware security exists [25]. They identify several challenges that lie with the automatic gathering and analysing of firmware, and describe how to tackle those challenges. In total the study looks at 32,356 firmware images, of any product they could find, and finds 693 images to be vulnerable. While the study is a quantitative analysis as opposed to our qualitative analysis, many of the challenges that are described are also present in this research. Apart from these challenges, they also find that 63% of the firmware images they found are for the ARM architecture, and 86% is built on top of Linux. Seeing as they target all types of firmware, and their data set is so large, this shows the relevance of the ARM architecture in the overall

picture.

Xu et al [26] make use of something they dubbed *logic bombs*, which are code snippets signifying specific challenges in symbolic execution such as detecting a stack overflow or dealing with file interaction. Using a set of these logic bombs, which they created, a benchmark for three symbolic execution frameworks (KLEE, Angr, and Triton, all described in section 3.3) is established. Since the logic bombs focus on 1 challenge each, and are very small compared to real world software, the study can accurately state which tool is proficient at which challenge. This work [26] differs from our work in the scope of the analysis, as it does not look at real world examples, or at techniques other than symbolic execution. So while our approach looks at which technique is better at finding certain types of vulnerabilities, [26] aims to reduce the problem of detecting vulnerabilities to small subproblems and compare the number of problems each symbolic execution implementation can solve. Trying to replicate this approach with challenges on multiple techniques would reiterate the fact that different techniques face different challenges, rather than give usable results on their performance.

A very theoretical approach to a comparison is [27], where the authors define a way to analyse taint analysis methods through the principle of non-interference. The rules of taint propagation that are used by each implementation are judged based on whether they are sound (void of false negatives) and complete (void of false positives). They then build their own taint analysis engine with rules that are found to be sound, and mostly complete. The authors compare their system with other implementations, and find many are lacking in both soundness and completeness. However, they also remark that no taint analysis can be fully sound and complete. A practical comparison is made with TEMU (see section 3.3.8), concluding that TEMU does not use completely sound rules. The study is a good example of a formal approach to the verification of the effectiveness of an analysis method. Our work differs from [27] in that it compares different analysis techniques rather than different taint analysis methods, and our approach is empirical whereas [27] is theoretical.

3.2 Performance metrics

It is straightforward to find possible advantages and disadvantages of vulnerability detection techniques, but how to measure them is not. The main metric used by studies is the number of vulnerabilities found (in e.g. [25, 26, 28]). An issue with this metric is the possibility of false positives. If a tool reports vulnerabilities where there are none it could be seen as better, while effectively it would be worse since an analyst would have to sift through more false information. When examining binaries with known vulnerabilities, however, a *ground truth* can be created, which can serve as a benchmark for each tool. A second metric most studies use is general to all computational problems, namely the running time. The running time usually is capped at some value, especially for symbolic execution, to stop them from running practically forever. When two tools find the same number of vulnerabilities in this time frame, but one does so faster than the other, it is considered better. Some studies also use a third metric, memory usage. A limited amount of memory can be an issue for both taint analysis [17] and symbolic execution [29], hence a tool that can give an upper bound to its memory usage has some advantage over a tool that can not. Memory usage has been left out of scope however, since it is non-trivial to measure for each separate technique in a framework, and thus would take away too much time from the main goal of the thesis, while adding very little information to other standard metrics such as time.

Different binary analysis techniques have different approaches to finding vulnerabilities, so it may not be surprising that one technique can be better at finding certain types of vulnerabilities than others. Taint analysis will always result in vulnerabilities related to user input, e.g. [30, 27]. Symbolic execution is likely more complete in its findings, such as being able to detect freeing a variable multiple times. Therefore an overview is needed, of vulnerability types found in binary code that are automatically detectable. Such an overview can be found in Table 1. For every type of vulnerability found in the literature, the relevancy is estimated based on the sources, and are listed together with the sources.

Vulnerability type	Relevancy	Source
Stack based buffer overflow	Very common	[30, 21, 28, 31, 32, 33]
Heap based buffer overflow	Common	[21, 31, 32, 33]
Format string	Common	[31, 32, 33]
NULL pointer dereference	Common	[31, 32, 33]
Integer overflow	Uncommon	[31, 33]
Use after free	Uncommon	[21, 31, 32, 33]
Encryption with insufficient entropy	Uncommon	[33]
Leaking memory	Uncommon	[33]
Race condition	Uncommon	[33]

Table 1: Vulnerabilities in the IoT related to the firmware itself.

3.3 Existing tools

This section gives an overview of the state of the art implementations of one or more vulnerability detection techniques discussed in the previous chapter. Some entries describe tools, e.g. a fuzzing tool like AFL, and some describe frameworks (e.g. Angr). Frameworks combine the properties of different techniques, or offer them as separate functionalities.

3.3.1 Angr

The Angr framework [34] stems from a state-of-knowledge type of research that resulted in the researchers creating their own platform for binary analysis. As such, it implements its own version of all described techniques. For CFG recovery it defines two algorithms, for an accurate and a fast approach, which can both be applied to get the most complete graph. The accurate algorithm combines the techniques of forced execution, backward slicing, symbolic execution, and value set analysis. This combination allows Angr to resolve many indirect control flow transfers. When a CFG has been recovered, Angr can refine any results using VSA into what the authors call a Value Flow Graph (VFG). This is an enhanced CFG with the program state as approximated by VSA at each instruction. Angr also uses two different types of symbolic execution: dynamic symbolic execution, and under-constrained symbolic execution. The first is based on the techniques described in Mayhem (see section 3.3.3). The second technique comes from KLEE (see section 3.3.2). Lastly it allows symbolic-assisted fuzzing, which combines the fuzzing of AFL (section 3.3.5) with the symbolic execution engine of Angr. The whole platform is architecture independent, building off libVEX, the IL lifter of the Valgrind project [10].

3.3.2 KLEE

KLEE [21] is a symbolic execution tool which, apart from variables in a program, allows an analyst to model a file system symbolically. The technique behind the symbolic execution was naive in early versions, but with later additions [20, 35] the running time and accuracy can be improved. KLEE essentially is a Virtual Machine (VM), running on top of LLVM. The base version of KLEE assumes access to the source code of a program for its analysis, however, so can not be used in our setting. An extension to the framework was made [36] that uses a combination of memory dumping, function hooking, and lifting code to make KLEE work on binary code. Since this work was published close to the deadline of this thesis, however, it is not included.

3.3.3 Mayhem

Mayhem [29] combines 'online' and 'offline' symbolic execution, calling the combination hybrid symbolic execution. Mayhem defines online execution as trying to execute all possible paths in a single run of the system, and offline execution as first running in a concrete mode and afterwards running symbolically, using the path of the first run. Offline execution here is the same as concolic execution. The authors try to combine these to avoid completely filling the memory and thereby not being able to explore all paths, while simultaneously being able to explore every path in the end. Together with this combination, the splitting and optimising of formulas, and a combination with taint analysis, Mayhem runs faster than previous analysis frameworks (according to the authors). Mayhem works on x86 and ARM architectures, as it builds off the Binary Analysis Platform (BAP) [37].

3.3.4 Triton

Triton [38] (paper in French) is an analysis framework providing symbolic execution and taint analysis. It builds on any binary instrumentation tool such as Intel Pintool [8], which is a binary instrumentation framework. It only supports the x86, x64, and AArch64 instruction sets however. Since our scope is limited to native ARM architectures and Thumb-2, this makes the framework as is less useful for our research.

3.3.5 AFL

AFL [14] is a type of mutational fuzzer, with the sole metric of performance being code coverage. While there are other mutational fuzzers, AFL stands out because it does not mutate only a single test case, and instead keeps all test mutations until it crashes the target. This allows it to find multiple distinct sets of inputs that each generate their own behaviour, rather than stopping after it finds a single crashing input. It stands out from other fuzzers as well because it only has the one heuristic for success, while still finding many crashes. While AFL as is only runs on source code, it is possible through the use of QEMU [39] to run it on binaries, regardless of original architecture. A fork project from AFL is AFL-Unicorn [15], which uses the mutation and instrumentation functionality from AFL in combination with the CPU emulation from Unicorn [16].

3.3.6 BARF

BARF [7] is a binary analysis framework, providing a translation from binary code to the Reverse Engineering Intermediate Language (REIL) [40], and an SMT solver used to simplify the code where possible. The authors offer this as a basis for the implementation of actual vulnerability finding methods, but do not provide any themselves. It supports various architectures through REIL, but since no other modules have been implemented, it is not sensible to compare it to other frameworks. It also can not process the Thumb-2 ISA, which leaves the framework out of our scope.

3.3.7 S2E

Selective Symbolic Execution (S^2E) [41] is a framework for symbolic execution. It distinguishes itself by making all values either concrete or symbolic, by selecting which type suits a variable better. The authors call this selective symbolic execution, which aims to speed up the analysis by keeping all variables that depend on known inputs concrete. All variables resulting from unknowns are made symbolic, such that constraints on them can be solved later. Conversion functions are defined between symbolic and concrete mode, which can (or should) be used when a variable is needed in one mode while being in the other. S^2E builds on QEMU [39] and KLEE [21], using

those as an emulator and symbolic execution engine respectively. Because of these dependencies, it should function on x86, x64, and ARM architectures. However, at the time of writing S^2E does not actually use QEMU for a full emulation, but only for the translation of instructions. The layout of the CPU is defined by S^2E itself, and at the time of writing the newer versions do not have a definition for ARM yet. The older versions do not use a wide spectrum of newer symbolic execution features, rendering them obsolete.

3.3.8 BitBlaze

BitBlaze [42] is a framework for binary analysis, and is split up into three components. The first one, Vine, is a static analysis module that is used to disassemble and lift a binary to the Vine IL, and then optimise the result. After the initial translation of binary to an optimised IL the module can be used to recover CFGs, resolve indirect control flows using VSA, or lift from the IL to a higher level language such as C. The second component is called TEMU, which is a dynamic analysis module based on QEMU [39]. This can be used to retrieve further information about the program, such as process, thread, or symbolic information. It also allows for taint analysis, which is implemented by BitBlaze itself with no notable details. Lastly, there is a symbolic execution module called Rudder, which runs on top of the TEMU module. Similarly to S^2E (see section 3.3.7) it runs symbolic execution partially concrete and partially symbolic, though unlike S^2E, BitBlaze does not explicitly define conversions between the two modes. It uses VSA to determine the set of addresses a symbolic pointer can point to, so this does not have to be an issue when analysing.

3.3.9 BAP

BAP is an analysis framework which builds off the static analysis part of Bitblaze (section 3.3.8) [37]. Since the IL used by Vine is based on VEX [10], did not have formal specifications, and did not support bi-endian architectures such as ARM, the authors decided to rewrite everything to work with their own IL, the BAP IL (BIL). It implements taint analysis [43] and symbolic execution. The BIL supports lifting from x86, x64, ARM, and MIPS. Lastly, it is open source and is being maintained. BAP thus is a good candidate for this work.

3.3.10 Manticore

Manticore is a project similar to Angr (section 3.3.1), and Triton (section 3.3.4). It supports symbolic execution, taint analysis, and fuzzing. According to their Github page[44], the framework supports ARM binaries. Due to lack of clear documentation, however, we were unable to find this functionality. Manticore builds off of Unicorn [16], which supports ARM, but Manticores own definition of a binary file seems to exclude binaries based on the ARM ISA. While the support may be buried somewhere in the framework, according to their own Github issues the ARM CPU definition is still lacking.

3.3.11 Driller

Driller is a fuzzing tool based on Angr (section 3.3.1) and AFL (section 3.3.5). It aims to fuzz until it detects it is stuck on continuously trying similar inputs, and then it uses the symbolic execution component from Angr to find more interesting inputs. Like AFL, it is driven by a code coverage metric. While Driller looks promising as an implementation, it can not currently be used to fuzz cross-platform.

3.3.12 Others

There are several other frameworks related to binary analysis, including but not limited to Avatar^2 [45], FACT, DART, Firmadyne, Valgrind, etc. Some of these do not give any implementation of fuzzing, taint analysis, or symbolic execution, and are therefore not interesting in the context of this thesis. Others are simply too old or have been superseded by newer frameworks. In the second case interesting concepts have often been used in other frameworks, but these will not be covered explicitly and instead referenced.

Chapter 4

Methodology

The previous chapter covered what the state of the art techniques and their implementations are. In order to answer the research question, there are three more thing to establish: the selection of firmware, performance dimensions of the analyses, and criteria of evaluation. In this chapter these topics are addressed, resulting in a systematic approach to the analysis.

4.1 Firmware

4.1.1 Firmware selection and measurement

The field of IoT is quite big, and firmware very diverse. As described in section 1.3, there are several ways to tone this down, but this does not quite deal with the issue of different attack vectors on different categories of firmware. For example, it is reasonable to expect different vulnerabilities in routers or switches in smart door locks, since the firmware operating the devices are very different in functionality and type of input. Therefore a comparison based on multiple categories a priori is unreasonable, and we limit our analysis to one category of firmware only. It is important that the category selected is of devices with plenty of firmware images with known vulnerabilities, in order to establish a *ground truth*. Possible categories in the IoT, derived from sources in literature, are listed in Table 2. The sources are listed behind the respective categories. Of these categories, we see that the greatest impact comes from 'Routers and switches', 'Power plant controllers / PLCs', and 'Medical devices'. Of these categories, most firmware images are available in the first category, taking from sources such as Shodan [46] and File Transfer Protocol (FTP) search engines like NAPALM [47] and Mamont [48]. Searching for router or switch vulnerabilities in the CVEs also yields plenty of vulnerabilities to build a ground truth [49].

Firmware category	Why relevant	Source
Routers and switches	They are everywhere High impact if compromised	[50, 51, 52]
Home entertainment	Compromise will disturb people Can be used in a botnet	[25, 50, 51]
Home automation	Compromise will disturb people Can be used in a botnet	[51, 53, 54]
IP cameras	Can be used to physically spy on people Can be used in a botnet	[25, 51]
Power plant controllers / PLCs	Very high impact if compromised Often lacking security updates	[25, 51, 55]
Whiteware	Can be used in a botnet	[51, 54]
Cars	High impact if compromised Most people own one	[51, 56, 54]
Medical devices	Very high impact if compromised	[51, 57]

Table 2: Categories of firmware targets, i.e. the types devices in the IoT

Apart from the general set of criteria for the analysis of binaries in section 1.3 and the selection of a firmware category, some classification of the firmware images in our data set is needed. The data set the techniques are compared on should be representative of real world router firmware, in order to find advantages and disadvantages that apply to the real world. Large differences between these firmware images should be avoided: running the same technique on a large and complex image as well as a small and simple image will yield predictable results. Such differences can e.g. be the size of the binary code (in the case of ARM, the lines of assembly code) [58], the code complexity [58], and the optimisation level it was compiled at [23]. Similar studies, e.g. [25, 28] analyse the firmware images instead of the analysis techniques, for which it suffices to only have a representative sample. While distinguishing the images by their optimisation level would give a good idea of the difficulty of analysing an image [23], this information is lost upon compilation of the binary and thus not usable. Similarly, the code complexity could tell something about the difficulty of an analysis. Apart from the Kolmogorov complexity mentioned in [58], the cyclomatic complexity [59] can be used to estimate the difficulty of an analysis. The first is infeasible however, and the second can be trumped by using the number of lines of code [60]. Using the number of lines of code as a complexity metric instead of the cyclomatic complexity is proven to be approximately equivalent [61]. On ARM finding the number of lines of code is not trivial, as in-line data is prevalent. Taking the result from disassembly, however, gives an accurate estimate of the number of lines of code.

4.1.2 Firmware collection

Based on the firmware category we selected, routers and switches, we can gather a list of CVEs by searching the National Vulnerability Database [62] for vulnerabilities per known router vendor. While there is a possibility to search per product, this is less feasible both because it is more likely we omit relevant products than it is that we omit a vendor, and because some CVEs are not properly linked to a product. This second case happens regularly when a vulnerability report has been sent to the vendor and only one of the products found to be vulnerable is included in the actual CVE. The resulting list can then be manually filtered to only include vulnerabilities related to binary code, so only vulnerabilities relevant to binary analysis are left. Based on the resulting CVEs we know which firmware images to collect. Gathering a sufficiently large number of firmware images from this list calls for automating the task. Costin et al. use Custom Search Engines (CSE) from Google to scrape info and gather firmware [25]. Chen et al. [28] build off Scrapy, writing their own spiders to gather info, and manually finding FTP servers from vendors, which they simply download everything from that could be a firmware image. These are the only large-scale firmware gatherers in the literature by our knowledge, and we have only been able to find the second method available publicly.

Having written custom spiders for various router vendors, however, shows that most vendors no longer host vulnerable versions of their firmware. Manually probing support sites for different countries show that some are still hosted somewhere, but most are difficult to find. Following the approach as described in [25] does not work for most images with known vulnerabilities either; as images are hosted on FTP servers that do not allow indexing, and are not always linked to from elsewhere on the web. Therefore the data set is built from manually found firmware images, and as a result relatively small in comparison to similar studies, such as [25, 28].

4.1.3 Firmware unpacking

Both works cited in the previous section also review some firmware unpacking methods, with [25] creating their own framework, and comparing it with some others such as Binwalk [63]. One other such tool was named Binary Analysis Tool (BAT), which has since been deprecated and replaced by Binary Analysis Next Generation (BANG) [64]. Firmadyne [28] simply states they built around Binwalk, extending the functionality where their needs required it. In Firmadyne, Binwalk performed fairly well. Since BANG built on Binwalk however, and has been updated more actively and recently, it seems like the better choice for the unpacking of firmware. From our

own experimenting it seems BANG can unpack most firmware, and where it gets stuck on parts of an image Binwalk can often unpack that single part. In order to unpack as many acquired images as possible, we thus use both tools.

4.2 Analysis tool selection

In order to get a relevant set of implementations, we pick those that 1) are able to perform either fuzzing, taint analysis, or symbolic execution; 2) are publicly available as source code; 3) are able to operate on binary code; 4) are being updated regularly; 5) can either disassemble and lift Native ARM and Thumb-2 instructions, or operate on an IL that either of those can be translated to. In table 3 these criteria are set out against the implementation described in section 3.3. The only tools that fit all of these criteria are Angr and BAP, but since Triton and AFL-Unicorn are very relevant in the binary analysis field and fit most of the criteria, they will also be included in our analysis.

	Open source	Works on binary code	Updated regularly	ARM compatible IL (Native ARM/Thumb-2)
Angr	✓	✓	✓	✓
KLEE	✓		✓	
Mayhem		✓	Not publicly	
Triton	✓	✓	✓	✓
AFL	✓		Only plugins	
AFL-Unicorn	✓	✓		✓
BARF	✓	✓		
S2E	✓	✓	✓	Only old versions
BitBlaze	✓	✓		
BAP	✓	✓	✓	✓
Manticore	✓	✓	✓	Only in documentation
Driller	✓	✓	✓	Prone to errors

Table 3: Features of the state of the art binary analysis tools

4.2.1 Performance metrics

Following from section 3.2, we have several vulnerability types that are automatically detectable by the selected tools and related to binary code. Based on these vulnerability types, we define the following categories of vulnerabilities: 1) buffer overflows; 2) format string attacks; 3) use after free/double free; 4) NULL pointer dereference. These vulnerabilities have been chosen for being detectable in an automatic way, and their relevancy in the IoT setting. The ground truth, which can be found in section 4.4.1, consists of firmware which has a known CVE in one of these categories. The performance of each analysis technique will be measured by the number of CVEs confirmed, and the running time of the analysis.

4.3 Confirming found vulnerabilities

Matching up any results from the analyses with the ground truth of CVEs can be done in two different ways. All CVEs have a short writeup, which can be used to identify the vulnerable program(s) in the firmware. This same writeup can usually also be used to find the specific location of the vulnerability, which is an address in the binary. In cases where it is not possible to retrieve the exact location of a vulnerability from the writeup, it is still possible to do so by

means of binary diffing. This method locates differences between two binary files, by matching up basic blocks from each input file. In this work we use r2diff from the radare2 framework [65].

Most tools return addresses as part of a finding, which can be compared to the location found through the CVE or binary diffing. An exact match between an address in a finding and the address from a CVE is not necessarily needed, as it is possible the framework yielding the finding has a slightly different definition of a vulnerability location. To ensure no correct findings are left out, this necessitates a broader definition of vulnerability location. To this end we distinguish three levels of vulnerability location: the exact address, the basic block it is in, and the function it is in. These addresses and ranges are recovered from the binaries under analysis using a Python script we wrote for Binary Ninja [66]. Binary Ninja takes care of all disassembly, and defining basic block and function boundaries. The script uses this information to determine which addresses output by the analyses are in the same basic block or function as the address from the CVE.

4.4 Experiment setup

Combining the pieces from the previous sections, we can create a design for the experiment to be run. The entire flow can be found in figure 2. First the National Vulnerability Database (NVD) is crawled for vulnerabilities in routers that belong to any of the four categories defined in the performance metrics in section 4.2.1. From these vulnerabilities we select those where the vulnerable version of the affected product can be found with the methods described in section 4.1.2. For any of the selected vulnerabilities we also retrieve the information from the corresponding vulnerability writeup. The selected firmware images are extracted according to the method described in section 4.1.3. If this is successful, we usually get a kernel, filesystem, and a bootloader. In these we then locate the vulnerable binary code, using the information from the vulnerability writeup. This vulnerable code is the input to the frameworks implementing each of the binary analysis techniques. The output gained from these frameworks is then manually analysed, again using the information from the vulnerability writeup, to compare their performance based on the performance metrics. This comparison is thus qualitative, based on human analyst expertise.

4.4.1 Ground truth

The ground truth resulting from the criteria in section 4.1 can be found in Table 4. The number of instructions of each program were counted using the angr framework. Categorising the different programs based on the number of instructions, we can clearly distinguish three small programs, three large programs, and one very large program, using boundaries at 10k and 100k instructions. Notably, there are no use after free/double free vulnerabilities, and only one format string vulnerability in the ground truth.

4.5 Methodological limitations

Comparing binary analyses in this way has several limitations. First and foremost is the data set used. Vendors make a good effort keeping the vulnerable versions of their firmware away from the public, by simply not hosting it, locking all firmware downloads behind a customer-only login, or by encrypting the firmware images. This severely limits the number of firmwares, and thus the number of CVEs we can check for. Second, the data the analyses actually run on, the binaries, are real world examples. In general, they tend to be large and complex, and analysing them for vulnerabilities thus is difficult. In practice this can partially be overcome through reverse engineering the application, expert use of the analysis frameworks, or both. Since a fair comparison of the frameworks requires using strategies provided by the developers of those frameworks, expert use is fairly limited in our case. And since the topic involves automatic analysis, reverse engineering any application is kept to a minimum, only being used to interpret results.

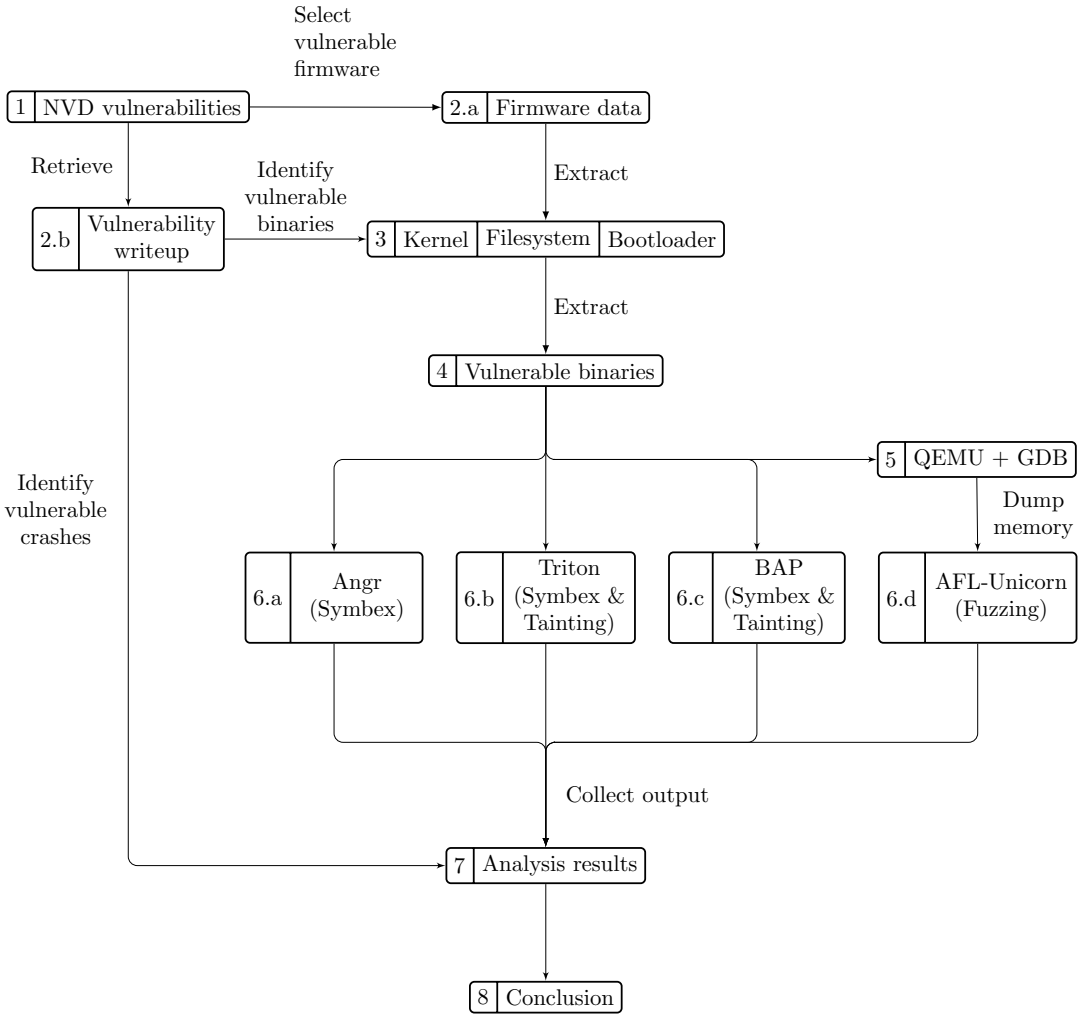

Figure 2: The setup of the experiment.

Product	CVE number	Vulnerability type	Program	#instructions	Category
EDR-810	CVE-2017-12124	Null pointer dereference	Webserver	387817	Very large
	CVE-2017-14435	Null pointer dereference	Webserver	387817	Very large
	CVE-2017-14436	Null pointer dereference	Webserver	387817	Very large
	CVE-2017-14437	Null pointer dereference	Webserver	387817	Very large
RouterOS	CVE-2018-1156	Buffer overflow	License upgrader	2066	Small
	CVE-2018-1159	Buffer overflow	Crypto library	2679	Small
	CVE-2018-7445	Buffer overflow	Samba	36088	Large
RT-AC3200	CVE-2018-14712	Buffer overflow	Disk utility library	2679	Small
	CVE-2018-14713	Format string	Web server	53344	Large
	CVE-2018-17127	Null pointer dereference	Web server	53344	Large
RT-AC5300	CVE-2018-17022	Stack buffer overflow	Web server	64370	Large
	CVE-2018-17127	Null pointer dereference	Web server	64370	Large

Table 4: The ground truth resulting from our criteria in section 4.1

Chapter 5

Results

5.1 Analysis results

To determine the performance of each analysis, they were run on an Intel Xeon E5-2697A v4 with a reported clock speed of 2.6 GHz (but measured clock speed of 5.2 gigahertz), on a virtual core, and 4040 megabyte virtual random access memory. For every framework the provided Python bindings where used. Running all analyses for a maximum duration of 20000 seconds (approximately five and a half hours) yields rough results as found in Table 5. All implementations yield different results, but all results contain an address. BAP's symbolic execution counts the number of incidents reported, which are violations of a set of heuristics in the framework meant to detect CWEs. It also inserts checkpoints and estimates the coverage of the analysis by counting these as they are encountered. BAP's taint analysis counts similar violations, the most prominent being a Control Flow Integrity (CFI) violation. Triton outputs the registers, memory locations, and instructions that are symbolic or tainted. However as Triton by default relies on Intel Pintool, which does not support the ARM architecture, it has been omitted from testing. While according to Triton's documentation it is possible to use a different tracer, we found no examples of how to do this. Angr keeps track of all symbolic states, and categorises errored or unconstrained states as such. AFL-Unicorn detects when a program crashes, and yields inputs causing a crash as result.

5.2 Qualitative analysis

Using the method described in section 4.3 we located the exact addresses, basic block ranges, and function ranges of each vulnerability location. These can be found in Table 7, in Appendix A. We compared addresses found by the analyses with these addresses and ranges, and the resulting matches can be found in Table 6. The only analyses to find any matches were the symbolic execution and taint analysis from BAP, and only on the binaries categorised as small. Since the results for both analyses are effectively identical, they are displayed together. Further analysis is done per binary, and detailed below.

5.2.1 License upgrader

Both BAP's taint analysis and symbolic execution have found 389 CFI violations, of which one is related to the information found in the CVE writeup. At the address tagged as violating the control flow is a *POP* instruction, which is an instruction that moves data from the stack into registers. One of those registers is the program counter, which keeps track of which instruction has to be executed next. This particular *POP* instruction is part of a function epilogue, and it makes sense it is flagged for a CFI violation as the corresponding CVE is a stack overflow. When the stack has been corrupted by an attacker, restoring the program counter sends the control to a location the attacker put on the stack. Further analysis shows that an earlier call to *sprintf* formats user-controlled strings without checking their length. A code snippet showing the vulnerable part of the application can found in Listing 2. The arguments to the *sprintf* function are stored in the registers, r0, r1, r2, and r3, and four more on the stack. From the function description we know that r0 holds a pointer to the resulting string, r1 holds a pointer to the format string, and all other arguments hold pointers to the data used in the formatting. From the binary we find that

	BAP (Symbex)	BAP (Taint analysis)	Angr (Symbex)	AFL (Fuzzing)
License upgrader				
Time (s)	125	86	20000	1
Result	389 incidents 100% coverage	389 CFI violations	No incidents	Crash due to threading
Crypto library				
Time (s)	626	292	20000	1
Result	274 incidents 100% coverage	274 CFI violations	No incidents	Crash due to unsupported instruction
Disk utility library				
Time (s)	974	483	20000	1
Result	1153 incidents 100% coverage	1153 CFI violations	2 unconstrained states	Crash due to a lack of process information
Samba				
Time (s)	20000	20000	191	1
Result	No incidents 7% coverage	3661 CFI violations	Crash due to Z3 running out of memory	Crash due to threading
Web server RT-AC3200				
Time (s)	20000	20000	4929	4
Result	No incidents 2% coverage	2 CFI violations	Crash due to Z3 running out of memory	Crash due to a lack of process information
Web server RT-AC5300				
Time (s)	20000	20000	3411	1
Result	No incidents 2% coverage	2 CFI violations	Crash due to Z3 running out of memory	Crash due to a lack of process information
Web server EDR-810				
Time (s)	145	146	155	-
Result	Crashes due to failed disassembly	Crashes due to failed disassembly	Crashes due to failed disassembly	Infeasible to create a memory dump

Table 5: Summarised results of the analyses.

	Exact matches	Basic block matches	Function matches
License upgrader	0	0	1
Crypto library	0	4	4
Disk utility library	0	0	20
Samba	0	0	0
Web server RT-AC3200	0	0	0
Web server RT-AC5300	0	0	0
Web server EDR-810	0	0	0

Table 6: Number of matches in the results from BAP's analyses.

the format string is:
GET /ssl_conn.php?usrname=%s&passwd=%s&softid=%s&level=%d&pay_type=%d&board=%d.
The pointers to the username and password thus must be stored in r2 and r3 respectively. While the result is clear enough to get an idea of what the vulnerability might be, there are 195 other such messages in the results.

```
ldr r3, [r4, #0x560]
ldr r1, 0xc467              // Pointer to the format string
mov r2, r7                  // Pointer to the username
add r3, r3, #0x4
stm sp, {r3, r9}
mov r3, #0x2
str r3, [sp, #0xc]
ldr r3, [sp, #0x20]         // Pointer to the password
add r0, sp, #0x38           // Pointer to the resulting string
str r8, [sp, #0x8]
add r3, r3, #0x4
bl sprintf                  // Call sprintf
...
pop {r4, ..., pc}           // Put values from the stack in
                               registers
```

Listing 2: Code snippet from the license upgrader, showing the buffer overflow.

5.2.2 Crypto library

Both BAP's taint analysis and symbolic execution have found 274 CFI violations, of which four are related to the information found in the CVE writeup. The address tagged for a control flow violation has a *LDRB* instruction, which copies one byte out of memory into a register. Tracing the stack usage back leads to other binaries in the firmware, a JavaScript proxy and the web server. According to the CVE writeup, when a user authenticates and disconnects in quick succession, the web server crashes. This happens at an out-of-bounds memory read in the crypto library. This can be seen in Listing 3. r0 here holds a pointer to a data structure used in RC4, including a pointer located at r0 + 0x100. This pointer should point to existing data, as it is dereferenced in the last instruction, but does not.

Trying to infer the type of the memory contents at the time of the out-of-bounds read proves

```
ldr          r2, [r0, #0x100]
ldr          r1, [r0, #0x104]
ldrb         r3, [r0, r2]     // Read from location r0 + r2
...
```

Listing 3: Code snippet from the crypto library, showing the out-of-bounds read.

to be difficult. A large struct with the context of the web server is stored here, part of which stores the data used for the RC4 cipher. The exact cause of this vulnerability was not traceable in a reasonable amount of time, even given the information provided in the CVE writeup, so with only the information provided in the output from BAP detecting the vulnerability would be very difficult.

5.2.3 Disk utility library

Both BAP's taint analysis and symbolic execution have found 483 CFI violations, of which 20 are related to the information found in the CVE writeup. Similarly to the license upgrade case at the address with the control flow violation there is a *POP* instruction. Also comparable to the license upgrader case, an analyst could trace back in the instruction and find a *sprintf* instruction that formats a user-controlled string without checking its length, thus resulting in the stack overflow in the CVE. A code snippet showing this can be found in Listing 4. The arguments here are similar to the disk library case; r0 contains a pointer to the result, r1 contains the format string, and the other arguments are pointers to the data used in the formatting. The format string here is */dev/%s*.
Since there is only one argument, we know r2 points to user-controlled data. Of the 1153 CFI violations, 1002 are about a function epilogue, giving results very similar to the license upgrader as well.

```
mov r2, r9                    // Pointer to the user controlled
                                 string
str r5, [r12, #0x14]
add r1, r4, r1                // Pointer to the format string
mov r0, r6                    // Pointer to the resulting string
bl  sprintf                   // Call sprintf
...
pop {r4, ..., pc}            // Put values from the stack in
                                 registers
```

Listing 4: Code snippet from the disk utility library, showing the buffer overflow.

5.3 Discussion

While implementations of both symbolic execution and taint analysis have given information confirming the CVEs of the small binaries, each also gave a flood of messages not related to the CVEs. We do not have a ground truth for these, nor is it in the scope of this thesis to further analyse the cause of all these messages, and thus we do not know whether these are of any added

value. If they are not, fewer messages would have been better as it would save a would-be analyst the time it costs to check them all. The running times are more or less expectable: the license upgrader has the fewest instructions and is analysed more quickly in both symbolic execution and taint analysis than the other small binaries. Likewise, taint analysis is faster than symbolic execution in cases where they both confirmed a CVE. Both taint analysis and symbolic execution can clearly be used to detect user input related vulnerabilities.

All of the analyses are dependent on some technology that makes them work in a practical setting: symbolic execution depends on the Z3 solver in order to not get stuck at variable branches, taint analysis depends on a way to trace the execution flow of the program, and fuzzing depends on an accurate emulator. In angr some crashes happened due to Z3 running out of memory, which is a common problem for all constraint solvers. In Triton the standard tracing framework is Intel Pintool, which does not work on ARM architectures, causing both Triton's symbolic execution and taint analysis to fail. The crypto library contained a supervisor call instruction which was unsupported by the Unicorn emulator, which caused AFL to crash. All analyses are also dependent on the first step of binary analysis as well; disassembly. A lot of the state of the art tools rely on the Capstone disassembler, and as such fail on any format or architecture that is not supported by Capstone. While generally this is not a problem since Capstone is built on top of LLVM, in the case of the EDR-810 this is actually why all analyses crashed. LLVM only supports the System V Applied Binary Interface (ABI), and the EDR-810 firmware was compiled for the GNU ABI.

The used dataset is too small to make make definite conclusions on which technique is more suitable, if any, to detect certain types of vulnerability. However with the data supplied we can conclude that BAP was the framework best suited in our case, but again too little data was available to say anything about vulnerabilities in ARM binaries in general. Based on the results we have, no advice can be given on which technique to use for which type of vulnerability.

As AFL-Unicorn is missing certain features needed for automation, it is not suitable for an automatic analysis. Due to this it failed on every binary. All cases needed functionality that was not available with the Unicorn emulator, or a specific input format that needed to be added manually. In the first case, the missing functionality such as threading could have been replaced with function hooks, which would prevent a thread from spawning and thus interfering with Unicorn, while setting variables such that the main program believes threading is working as expected. The input format could be detected automatically, but at the time of writing this is not included in AFL-Unicorn yet. This input format could also be retrieved manually through reverse engineering, but that would defeat the purpose of automatic analysis.

In order to circumvent the lack of data, a set of binaries could be compiled from sources similar to router firmware. For example using Opengear's router firmware development kit [67] and injecting vulnerabilities in the source code would yield router firmware with known vulnerabilities. Alternatively, open source router firmware such as OpenWRT [68] could be used. Doing so would avoid collecting CVEs and firmware images, but risks creating unrealistic vulnerabilities. For example, if all vulnerabilities in the created ground truth were due to user input related library functions the performance of some frameworks could be better than other. Taint analysis with those functions defined as a taint source can reasonably be assumed to find all vulnerabilities. In reality vulnerabilities unrelated to library functions exist, such as in a hand written string copy, so taint analysis could seem more complete than it is.

The aim of this work is to find advantages and disadvantages of automatic binary analysis techniques, to ultimately assist in improving binary analysis as a whole. Knowing these advantages and disadvantages could help in the further development of the techniques, for example by automatically resolving some of the problems faced by AFL-Unicorn. While this would ease the work of binary analysts trying to improve the security of devices, it would also do so for malicious actors trying to break into those very same devices. When automated binary analysis and exploitation eventually becomes advanced enough to no longer require expertise, it is quite imaginable that a larger portion of this group of malicious actors will be able to exploit software without its source code. In any case this would lower the bar for such actors, making it more appealing to exploit binary code.

Chapter 6

Conclusions

The ground truth contained too few CVEs in order to draw any conclusion regarding the trade-offs of the binary analysis techniques. Since the fuzzing technique did not work properly automatically, nothing can be concluded about it. As discussed in the previous chapter, taint analysis does finish faster than symbolic execution, but since there are so few CVEs we can not confirm whether this speed is traded for accuracy. It is also not clear whether this shorter analysis time is general to all ARM binaries.

A comparison between techniques is less sensible than we initially thought, for two reasons. The first being that the state of the art of binary analysis is moving towards combining techniques such that they can mitigate each others downsides. If the best approach to binary analysis involves combining techniques, knowing which techniques perform better separately matters little. The second reason is such a comparison is almost guaranteed to run afoul of implementation details. Because of the nature of the binary analysis problem, any implementation needs to have definitions for things such as the CPU layout of various architectures or the calling convention of a certain OS. This means any of those definitions can be missing, or otherwise limit the applicability of the implementation. It also means if one technique outperforms other techniques this is not necessarily due to the effectiveness of the technique, but might be due to the correctness of the implementation.

6.1 Future work

As noted in the limitations in section 4.5, binary analysis is an expertise driven technology. While some tools and frameworks are available to automate it, it works better when preceding any analysis with an information gathering phase, which ideally is partially manual. This is especially the case with fuzzing, since it needs to successfully (i.e. with no crashes) finish running the binary under analysis to check whether its instrumentation works as expected. In order to accomplish this analysts generally start by finding out what input the application needs to run normally. Future work could improve on this by automatically detecting valid input for the application, which in turn can also be used as an input for the fuzzer to mutate.

A second improvement could, and in most practical scenarios does, come from the combination of techniques. Triton has an option to use a fast taint analysis to determine what parts of an application to use a slow symbolic execution on. Driller, an extension on angr, alternates fuzzing and symbolic execution to continue analysing a binary even when one of the techniques looks like has stopped making progress. A combination of slicing and fuzzing is also possible. Here an analyst could create forward slices of all user input, and fuzz those slices instead of the entire binary.

References

[1] DA Andriesse. *Analyzing and securing binaries through static disassembly*. PhD thesis, Vrije Universiteit Amsterdam, 2017.

[2] Anh Quynh Nguyen. Capstone engine. Github Repository, 2014. `https://github.com/aquynh/capstone` (Accessed: 2019-02-12).

[3] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[4] Philip Koopman, Jennifer Black, and Theresa Maxino. Deeply embedded survivability. February 2007.

[5] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[6] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries, 2010.

[7] Christian Heitman and Iván Arce. BARF : A multiplatform open source binary analysis and reverse engineering framework, 2014.

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[9] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[10] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[11] Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 16–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[13] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[14] Michał Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/` (Accessed: 2019-02-19).

[15] Nathan Voss. Afl-unicorn. Github Repository, 2017. `https://github.com/Battelle/afl-unicorn` (Accessed: 2019-04-12).

[16] Anh Quynh Nguyen and Hoang Yu Dang. Unicorn. Github Repository, 2015. `https://github.com/unicorn-engine/unicorn` (Accessed: 2019-03-18).

[17] DA Andriesse. *Practical Binary Analysis*, chapter 10. No starch press, October 2018.

[18] DA Andriesse. *Practical Binary Analysis*, chapter 12. No starch press, October 2018.

[19] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 03 2008.

[20] Suhabe Bugrara and Dawson Engler. Redundant state detection for dynamic symbolic execution. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 199–211, San Jose, CA, 2013. USENIX.

[21] 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, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[22] Roberto Baldoni, Emilio Coppa, Daniele Cono Delia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[23] R. J. Rodriguez, J. A. Artal, and J. Merseguer. Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions*, 12(8):1572–1580, Dec 2014.

[24] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, 2018.

[25] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.

[26] H. Xu, Z. Zhao, Y. Zhou, and M. R. Lyu. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[27] Lok Kwong Yan and Heng Yin. SoK: On the soundness and precision of dynamic taint analysis. 2017.

[28] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, pages 1–16, 2016.

[29] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[31] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387. IEEE, 2005.

[32] Yue Chen. *Securing Systems by Vulnerability Mitigation and Adaptive Live Patching.* PhD thesis, The Florida State University, 2018.

[33] Steve Christey and Robert A Martin. Vulnerability type distributions in CVE, 2007.

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

[35] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., 2015. USENIX Association.

[36] Sai Vegasena. Klee-native. `https://github.com/trailofbits/klee`.

[37] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[39] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[40] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[41] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number CONF, 2009.

[42] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[43] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[44] Trail of Bits. Manticore. Github Repository, 2017. `https://github.com/trailofbits/manticore` (Accessed: 2019-03-18).

[45] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)(February 2018), BAR*, volume 18, 2018.

[46] John Matherly. Shodan. `https://www.shodan.io/` (Accessed: 2019-04-08).

[47] Napalm. `https://www.searchftps.net/` (Accessed: 2019-04-17).

[48] Constantine Aygi. Mamont. `http://www.mmnt.ru/int/` (Accessed: 2019-04-17).

[49] Mitre. Commmon Vulnerabilities and Exposures. `https://cve.mitre.org/index.html` (Accessed: 2019-04-08).

[50] 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*, pages 437–448. ACM, 2016.

[51] Jonas Zaddach and Andrei Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.

[52] Craig Heffner. How to hack millions of routers. *Blackhat USA*, 1, 2010.

[53] Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin. Smart nest thermostat: A smart spy in your home. *Black Hat USA*, pages 1–8, 2014.

[54] Alan Grau. Can you trust your fridge? *IEEE Spectrum*, 52(3):50–56, 2015.

[55] Gaoqi Liang, Steven R Weller, Junhua Zhao, Fengji Luo, and Zhao Yang Dong. The 2015 ukraine blackout: Implications for false data injection attacks. *IEEE Transactions on Power Systems*, 32(4):3317–3318, 2017.

[56] P. Kleberger, T. Olovsson, and E. Jonsson. Security aspects of the in-vehicle network in the connected car. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 528–533, June 2011.

[57] Steve Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Jeremiah Blocki, Kevin Fu, and Dawn Song. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In *HealthSec*, 2011.

[58] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Towards automated classification of firmware images and identification of embedded devices. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 233–247. Springer, 2017.

[59] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[60] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[61] Graylin Jay, Joanne E Hale, Randy K Smith, David P Hale, Nicholas A Kraft, and Charles Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009.

[62] NIST Computer Security Division. National vulnerability database. `https://nvd.nist.gov/`.

[63] Craig Heffner. Binwalk. `https://github.com/ReFirmLabs/binwalk`.

[64] Armijn Hemel. Bang. `https://github.com/armijnhemel/binaryanalysis-ng`.

[65] Sergi lvarez. Radare2. `https://github.com/radareorg/radare2`.

[66] Vector 35. Binary ninja. `https://binary.ninja/` (Accessed: 2019-04-15).

[67] Opengear. Opengear custom development kit. `https://ftp.opengear.com/download/manual/current/opengear-custom-development-kit-user-guide.pdf`.

[68] OpenWrt. OpenWrt. `https://openwrt.org/`.

Appendix A

CVE locations

Product	CVE number	Exact address	Start basic block address	End basic block address	Start function address	End function address
EDR-810	CVE-2017-12124	0x61460	0x61460	0x6146c	0x6143c	0x618e4
	CVE-2017-14435	0x1b544	0x1b544	0x1b558	0x1b310	0x1c4ac
	CVE-2017-14436	0x1b55c	0x1b55c	0x1b570	0x1b310	0x1c4ac
	CVE-2017-14437	0x1b574	0x1b574	0x1b588	0x1b310	0x1c4ac
RouterOS	CVE-2018-1156	0xbc88	0xbc24	0xbca8	0xbadc	0xbe94
	CVE-2018-1159	0x33ec	0x33e4	0x3420	0x33e4	0x3420
	CVE-2018-7445	0x10da8	0x10d9c	0x10dbc	0x10d6c	0x10de4
RT-AC3200	CVE-2018-14712	0x6b58	0x6b28	0x6b84	0x68dc	0x6c44
	CVE-2018-14713	0x13c48	0x13c40	0x13c54	0x13be0	0x13c60
	CVE-2018-17127	0x2e670	0x2e5b8	0x2e690	0x2e5b8	0x2e914
RT-AC5300	CVE-2018-17022	0x185f8	0x185ac	0x18604	0x18558	0x18864
	CVE-2018-17127	0x340a4	0x33fec	0x340c4	0x33fec	0x34348

Table 7: The exact addresses and ranges for each CVE in each product.