# Dynamic binary firmware analysis : challenges & solutions

Marius Muench



**THESE DE DOCTORAT DE**
**SORBONNE UNIVERSITE**

Spécialité « Informatique »

(Ecole doctorale)

Présentée par

Marius Muench

Pour obtenir le grade de

**DOCTEUR de SORBONNE UNIVERSITE**

Sujet de la thèse :

Dynamic Binary Firmware Analysis: Challenges & Solutions

soutenue le 19.09.2019

devant le jury composé de:

| | |
|---|---|
| M. Directeur de thèse | Davide BALZAROTTI |
| M. Co-Directeur de thèse | Aurélien FRANCILLON |
| M. Rapporteurs | Jean-Pierre SEIFERT<br>Roberto DI PIETRO |
| M. Examinateurs | Mathias PAYER<br>Sarah ZENNOU<br>Marc DACIER |

# Preface

I deeply want to thank my family, friends, colleagues, and companions I met along the way. This thesis is the result of all the aid, support, and guidance I received from you, as well as all the good hours and fun moments we spent together. But instead of writing down a list, in which I either forget someone or have to exclude people for space limitations, I rather want to use this opportunity to direct words of sincere concern to another person: The reader of this document.

This world needs more positivity. We live in times of hate, fear, and uncertainty and while we humans are busy fighting each other, our planet is slowly dying. There is no magic bullet to solve all problems at once and we can certainly not make every battle to our own.

What we can do, though, is being excellent to each other. Reach out to the ones who are dear to you and tell them that you love them. Be forgiving to people who were unjust in the past, for they may have changed. And, most importantly, engage with persons you never met. Share some knowledge, provide help where needed, or just be the cause for a smile. Overall, it is the sum of little actions which can improve humanity; not only science is standing on the shoulder of giants.

I may not know you, but if you help making this world a better place, you shall have my gratitude. The following words are worn out, but may have never been more true: We only have one life. Let's make it count.

# Abstract

Embedded systems are a key component of modern life and their growing interconnectivity makes their security a concern of utmost importance. Hence, the code running on those systems, called "firmware", has to be carefully evaluated and tested to minimize the risks accompanying the ever-growing deployment of embedded systems.

One common way to evaluate the security of firmware, especially in the absence of source code, is dynamic analysis, which requires the firmware code to be executed, either on the physical device, or inside an emulator. Unfortunately, compared to analysis and testing on desktop systems, dynamic analysis for firmware is lacking behind. In this thesis, we identify five main challenges preventing dynamic analysis and testing techniques from reaching their full potential on firmware: firmware retrieval, platform variety, fault detection, scalability, and instrumentation.

Through this thesis, we point out that rehosting is a promising approach to tackle these problems and develop `avatar`$^2$, a multi-target orchestration framework which is capable of running firmware in both fully, and partially emulated settings.

Using this framework, we adapt several dynamic analysis techniques to successfully operate on binary firmware. In detail we use its scriptability to easily replicate a previous study, we demonstrate that it allows to record and replay the execution of an embedded system, and implement heuristics for better fault detection as run-time monitors. Additionally, the framework serves as a building block for an experimental evaluation of fuzz testing on embedded systems, and is used in a scalable concolic execution engine for firmware.

Last but not least, we present Groundhogger, a novel approach for unpacking embedded devices' firmware which, unlike other unpacking tools, uses dynamic analysis to create unpackers and we evaluate it against three real world devices.

# Contents

# Chapter 1

# Introduction

Modern life is increasingly depending on computing systems of all sorts. These systems range from low end embedded devices to personal computers and large cloud-based system. They are deployed nearly everywhere from consumer electronics to safety-critical systems such as medical devices, Industrial Control Systems (ICS), autonomous vehicles, and home automation.

A particularly more and more important key component of this development is the evergrowing deployment of embedded systems. On the one hand, these systems are a fundamental building block for more powerful systems, like the network interface controller inside a commodity PC or the baseboard management controller inside a server. On the other hand, interconnected embedded devices and associated online services are the bread and butter of the so called "Internet of Things" and the amount of deployed devices is increasing day-by-day. While it is impossible to accurately determine the precise number of active connected devices worldwide, it is commonly agreed that this number is continuously growing and likely to exceed 20 billion by 2020 [Nor16].

Although the interconnectivity, complexity, hardware, and field of application vary highly among those devices, they all need to run software to some extent. The software specifically tailored for embedded devices is commonly referred to as *firmware*, and differs from traditional software by directly controlling the underlying hardware, which interacts with the outside world in some way.

Unfortunately, like other software, firmware is seldom bug-free, especially as significant parts of firmware are typically programmed in unsafe, low-level languages to facilitate hardware control. Even worse, programming errors in these languages frequently subvert the security of a device, opening the door for potential at-

tackers. Indeed, instances of exploited vulnerabilities in firmware are plentiful [Wei12, Gal17, Art17, CZF16, CWBE16, PGC18], which indicates that the rapid growth of connected devices is accompanied by an increase of attack surface.

This is especially daunting as many devices' applications not only span consumer electronics but also aforementioned safety-critical systems. Although the security of those systems is particularly vital, recent attacks are drawing a devastating picture of reality: Malware like Mirai was attributed to puppeteer more than tens of millions devices in 2016 [LN17], subsystems in vehicles of notable manufactures [MV15, Kee16] as well as third party devices [FPKS15] have been proven to be vulnerable to remote attacks, and more than 1200 attacks on ICS have been observed in 2015 alone [McM15].

Summarizing these attacks it becomes evident that the insecurity of embedded devices and their firmware pose a severe threat to our physical safety and the proper functioning of current and future societies. Hence, it is inevitable that assessing, evaluating, and improving the security of embedded devices is a task of utmost necessity, not only for vendors, but also by third party companies and security researchers.

While automated security testing for traditional software running on home and server computing systems is common praxis nowadays, and even spawned a full industry on its own, testing of firmware is still mostly carried out in a manual device to device ad hoc manner, if at all. Unsurprisingly, tools and techniques coping with the unique challenges presented by embedded systems are still in their infancy and lack behind their according counterparts for traditional software. Lack of transparency and the sheer diversity given by the devices tremendously complicates the adaptation of even simple software analysis techniques to firmware. To make matters even worse, source code is most of the times only available for manufacturers, while retrieving the firmware—even in binary format—from a given device for analysis is often a challenge on its own for third party testers.

## 1.1 Problem Statement

Firmware for embedded devices frequently contains security critical bugs, which can be uncovered using well established dynamic program analysis and testing techniques. However, to apply according analysis and testing techniques it is first necessary to identify, understand, and overcome the unique challenges posed by embedded devices.

On closer examination, many vulnerabilities found today on embedded devices can still be considered as "low-hanging fruits", such as weak authentication, insecure default configuration, hardcoded credentials, or unauthenticated management

interfaces [ALAM19]. While these types of vulnerabilities could be easily mitigated by enhancing the awareness of both vendors and end-users, another class of vulnerabilities are haunting the firmware of those devices and traditional software alike: memory corruptions caused by programming errors.

Automated software testing techniques such as fuzzing or symbolic execution are evermore popular and a key component for uncovering these programming errors. Unfortunately, when comparing to desktop and server systems, the adaptation of these techniques to embedded systems is significantly lacking behind. Given an arbitrary device, what are the hindering factors for applying state-of-the-art dynamic testing techniques, and how can they be overcome?

This thesis aims to answer this question, and revolves around testing the firmware of embedded systems for potential security vulnerabilities, while restricting its scope to *binary* firmware. We chose this approach because third party analysts rarely have access to the source code of the firmware, and even if source code is present, low-level hardware interaction is oftentimes implemented in assembly which maps directly to the binary representation.

## 1.2 Contributions

In this thesis, we tackle the challenges of binary firmware analysis and testing systematically. For doing so, we first pinpoint the core issues preventing effective and generic analysis techniques to be deployed: (1) firmware retrieval, (2) platform variety, (3) fault detection, (4) scalability, and (5) instrumentation. We provide a security-centric classification scheme for embedded systems and measure the effects of lacking fault detection mechanisms across different devices.

To overcome the issues themselves, we focus on rehosting based approaches and present `avatar`$^2$, a flexible multi-target orchestration framework suited to enable a variety of dynamic binary analysis techniques for firmware. The framework allows for partial emulation and provides peripheral modeling capabilities to overcome platform variety.

Using this framework, we replicate prior research and demonstrate the viability of record and replay for firmware execution which allow for additional instrumentation. Furthermore, we implemented a basic set of fault detection heuristics on top of the framework, and use those to evaluate different approaches to fuzz testing firmware. Additionally, we present a prototype for scalable concolic execution of firmware. Last but not least, we take on the challenge of firmware retrieval and present a novel approach to firmware unpacking based on the record and replay primitives provided by `avatar`$^2$ before highlighting future research directions and concluding the thesis.

## 1.3 Organization of this Thesis

In summary, this thesis is organized in 7 chapters and the next chapter will provide the necessary background for the rest of this work.

The following Chapter 3 will outline the challenges to dynamic firmware analysis and testing and present our classification scheme for embedded devices. Furthermore, we will demonstrate the severity of memory corruptions for typical configurations of embedded systems and discuss different approaches suited for improving dynamic firmware analysis.

The result of this discussion indicates that rehosting based solutions are most promising and Chapter 4 elaborates on rehosting as a strategy to tackle firmware analysis. Previous work on the topic is highlighted first, together with the drawbacks and advantages of the particular solutions. Furthermore, we present `avatar`$^2$, a multi-target orchestration system we developed and which allows for flexible rehosting in different scenarios.

Chapter 5 showcases our contributions to dynamic binary firmware analysis enabled by the framework in detail. Additionally, a set of fuzzing experiments, and a prototype for scalable concolic execution of firmware are presented.

Afterwards, Chapter 6 sheds not only additional light on the fundamental issue of firmware retrieval but also presents *Groundhogger*, a tool implementing a novel approach to firmware unpacking.

Eventually, we conclude the thesis with Chapter 7 and give an outlook on the future of dynamic binary firmware analysis and a summary of how we tackled the challenges of dynamic binary firmware analysis.

## 1.4 Publications & Open Source Releases

This thesis is based on two publications and two ongoing projects. The paper "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices" [MSK+18] and "Avatar$^2$: A Multi-target Orchestration Platform" [MNFB18] build the foundations of this thesis. While significant parts of these publications are used throughout the thesis, the structure of the thesis does not map directly to them. Specifically, the core parts of the first publication are reflected in Chapter 2 and 3, as well as Section 5.3 and 5.4. The essence of the second one can be found in Chapter 4, as well as in Section 5.1 and 5.2.

The first ongoing project with the working title “Groundhogger” is under finalization at the time of writing and Chapter 6 find its roots in this work. The second ongoing project incorporated in this thesis can be considered to be rather work-in-

progress. It is provisionally named “Terrace“ and preliminary results are displayed in Section 5.5.

The authors of this thesis sincerely believe in Open Source not only as cornerstone of open science, but also as crucial artifact for replicable research. Naturally, various open source releases are coupled with this thesis. First and foremost, the full `avatar`$^2$-framework which builds the foundation to major parts of the presented work is available at: https://github.com/avatartwo/avatar2.

Furthermore, both above mentioned publications are accompanied by additional releases for automatically setting up environments ready to run the presented experiments. These releases can be found at https://github.com/avatartwo/ndss18_wycinwyc and https://github.com/avatartwo/bar18_avatar2.

Last but not least, we want to assure that we are committed to make the implementations for both “Groundhogger” and “Terrace” openly available in a timely manner.

# Chapter 2

# Background

In this chapter, we provide the necessary background knowledge for the remainder of the thesis. Particularly, we give a more detailed definition of *what* an embedded system is, establish a clear distinction between bugs, crashes, vulnerabilities, and corruptions, and present important dynamic binary analysis techniques in the context of this thesis.

## 2.1 Embedded Systems

Embedded systems have pervaded modern life to an unprecedented scale. For example, they are the core of various commercial off-the-shelf (COTS) devices such as printers, mobile phones, home appliances, and computer components and peripherals. They are also present in many devices that are less consumer oriented such as video surveillance systems, medical implants, automotive elements, military systems, supervisory control and data acquisition (SCADA) and Programmable Logic Controller (PLC) devices, and basically anything the general public usually calls “electronics”.

However, a precise and general definition of what is an embedded system is hard to establish [Hea02] as they vary wildly in terms of hardware, computing power, purpose, and costs. In this thesis, we adopt the widely accepted idea that embedded devices are separated from modern general-purpose computers by two common characteristics: a) embedded systems are designed to fulfill a *special purpose*, and b) embedded systems often interact with the physical world through a number of *peripherals* connected to sensors and actuators.

While we will develop a more elaborate classification scheme for embedded systems themselves in Section 3.2, it is important to note that they can either be self-

contained or consist of several *embedded devices*. For the sake of simplicity, we will use the term embedded systems analogously to embedded device, as our work focuses on single devices, rather than on interconnected systems.

### 2.1.1 Firmware

Like traditional computing systems, embedded systems are driven by software to execute specified tasks. The software for embedded systems is called *firmware* and differs from traditional software in three points:

1. Firstly, firmware is typically directly interacting with the underlying hardware of the system to carry out its task. Traditional software, in comparison, frequently makes use of abstractions provided by the used language, system libraries, or the target operating systems.

2. Secondly, firmware is most of the time deeply integrated onto the embedded system and stored in read-only memory (ROM) or non-volatile memory chips such as flash memory or electrically erasable programmable read-only memory (EEPROM). Hence, firmware is installed onto the device by the manufacturer, rather than the user.

3. Lastly, well-defined executable formats as seen on desktop systems are the exception, and firmware commonly comes in a single "blob" or "image", which contains everything which is needed to ensure the operation of the device. Hence, data, code, and metadata are interleaved and the entrypoint for execution may be hardcoded inside the firmware to be directly used by the system's processor.

As we will show in Chapter 3, all three of this differences tremendously complicate applying existing binary analysis techniques to firmware.

### 2.1.2 Peripherals

An embedded system consists of at least one Central Processing Unit (CPU) to execute its firmware, and a set of peripherals. These peripherals are responsible for all sorts of input/output and general interaction with the physical world.

Nowadays, manufacturers frequently combine the processing unit with a set of predefined peripherals on a System-on-Chip (SoC); in those cases, one distinguishes frequently between *on-chip* and *off-chip* peripherals. This differentiation is necessary because the CPU can only interact with off-chip peripherals through special on-chip peripherals, such as Universal Synchronous/Asynchronous Receiver/-Transmitter (USART), bus, or network interfaces. The way interaction between

CPU and on-chip peripherals can be carried out is dependent on the CPU itself, but usually falls in one of the following categories:

- **Memory-Mapped Input/Output (MMIO)** requires that the hardware registers of a peripheral are mapped into the memory space accessible by the CPU. Hence, the peripheral's status, its configuration, and incoming and outgoing data, can be queried and set by reading from and writing to special memory locations.
- **Port-Mapped Input/Output (PMIO)** is used by some Instruction Set Architecture (ISA) families and introduces special instructions, such as `in` and `out` to communicate with the peripherals. In this case, the peripheral's registers are not mapped into main memory, but accessed via ports queried with those instructions.
- **Interrupt Requests (IRQs)** are initiated by the peripheral and usually notify the CPU about the occurrence of some sort of event to be processed, like the availability of new data, the completion of a timer, or even that a button was pressed. Upon arrival of an IRQ, the processor saves its current state at the soonest possible point and transfers the execution to the corresponding interrupt service routine (ISR) (also called "interrupt handler"). ISRs are designed to be short and once completed, the processor state is restored and execution continues from the previous point.

  Modern CPUs oftentimes implement a complex interrupt controller, which for instance allow nesting of interrupts, assigning priorities to them, and selectively disabling and enabling them.
- **Direct Memory Access (DMA)** requires the presence of a dedicated DMA controller, a specialized hardware component and peripheral on its own, which is capable of transferring data between other peripherals and main memory independent of the CPU. This allows for the exchange of large amount of data while the processor can execute other tasks. In most cases, the DMA controller notifies the CPU about a completed data transfer by issuing an interrupt.

## 2.2 Bugs, Faults, Corruptions & Crashes

The main motivation for improving dynamic binary firmware analysis is to uncover bugs with potential impact on a device's security. Therefore, it is vital to understand the relationship between bugs, faults, corruptions, and crashes. Generally speaking, both firmware and software are built to solve specific tasks, and thus

they follow a—not necessarily explicit—specification of what they are intended to do. It is also well known that software and firmware code can contain bugs: errors that can bring a program into an unintended state. When such an unintended state can be exploited by an attacker, a bug is classified as a security vulnerability.

A common class of bugs that often leads to vulnerabilities are memory corruptions or memory errors. More precise, *spatial memory errors* are out-of-bounds accesses of a memory object, whereas *temporal memory errors* represent accesses to a memory object that does not exist anymore [SPWS13].

A memory corruption itself can cause an *observable crash* of a program, whereby the program is either terminated or some recovery procedures, such as exception handlers, are executed. Today, many common security mechanisms – such as stack canaries or heap consistency checks – trigger these crashes. However, under particular conditions, we can encounter memory corruptions which do not lead to any observable and immediate crash of the system. We will refer to these cases as *silent memory corruptions*. During a silent memory corruption the program continues its execution and enters an unintended *faulty state*[1].

The important consequence of silent memory corruptions is that the actual fault might only become noticeable at a later point in time when a certain functionality is requested or when a particular sequence of events is received. While this may not be considered a problem as long as the device continues to execute, it poses a significant threat for the safety and security of embedded systems. In fact, as soon as the system enters a faulty state, the integrity of its operation cannot be guaranteed anymore, and wrong data could be returned or processed at any time.

While desktop systems are also subject to silent corruptions, those are a lot less frequent because these systems deploy several lines of defenses. Although those lines of defenses are often designed for hardening programs against attacks, they also make faults more likely to lead to a crash. Those mechanisms include memory isolation, protection mechanisms, and integrity checks for memory structures.

Embedded systems, on the other hand, often lack similar mechanisms. This can not only lead to devastating consequences as soon as a system interacts with the exterior world, but it also complicates the security analysis of those systems as many black box testing techniques, and fuzzing in particular, rely on observable effects to infer the state of the device.

[1] Our definition for faulty states deliberately follows the definition of weird machine states proposed by Bratus et al. [BLP+11]. However, while the focus of weird machines is to provide a concept of exploitability in general, our definition only focuses on crashes caused by memory corruptions.

## 2.3 Binary Analysis & Testing

Binary-only testing is a very popular option even when source code is available to the analyst or when the software is available as open source. This is the case as some subtle bugs cannot be discovered by source code inspection only and because binary analysis allows testing of software independently of the source code language that was used to develop the system. Moreover, not relying on the source code means the analysis does not depend on the compiler to be correct, i.e., "what you fuzz is what you ship" [BGM13]. As a result, over the past decade, testing and analyzing software with binary-only approaches has become more and more popular and a variety of different techniques have been published for binary analysis [SWS+16].

Unfortunately, most of those techniques were designed and implemented for desktop systems, and can not be adapted easily to embedded devices. However, before we describe the challenges of dynamic analysis for firmware, we want to answer the question of why we focus on dynamic analysis and introduce key analysis and testing techniques in the remainder of this chapter.

### 2.3.1 Static vs Dynamic Analysis

Program analysis is commonly divided in *static* and *dynamic* analysis. Static analysis reasons about the program just by examining its code, whereas dynamic analysis relies on the actual execution of the program. Therefore, static analysis can provide sound results by analyzing all possible execution path of a program.

However, with static analysis the whole run-time state of the program is not available (e.g., heap structures, register contents or threading) and approximations are often needed to decide otherwise undecidable problems. This is problematic due to two reasons: (a) the approximations may easily lead to a high number of false positives, and (b) in the context of firmware analysis, deciding where to deploy the approximations becomes a challenge on its own, as commonly used abstractions provided by an operating system or system library are often not available.

Dynamic analysis avoids those problems by analyzing the actual behavior of a program while it executes on a given input (at the price of being able to observe only a small portion of the program state and code). This approach shines especially when applied for security testing and evaluation with the aim to discover new bugs that impact the security of the program under analysis: every found bug is a bug. Common examples of vulnerabilities arising from those bugs are authentication bypasses, memory corruptions, or memory disclosures, which can all be beneficial to a potential attacker.

In this context, binary program analysis, which is based on machine code only, is especially important for two reasons. First, binary code is the most accurate representation of the program as it is the code that is executed directly on the processor. Second, despite the existence of a steadily growing open source movement, a large number of programs are distributed only in binary form. This applies especially to programs written in memory-unsafe languages, such as C and C++, which are especially dominant among firmware programmers as they allow low-level access to the hardware.

### 2.3.2 Instrumentation & Sanitizers

Most dynamic analysis approaches make use of instrumentation in one way or another to collect additional information about a program or alter its functionality during runtime. Generally speaking, the process of instrumentation means to modify the software under analysis or its execution environment to incorporate the *instrumentation code*. More specifically, *static* instrumentation is the modification of a program before it is executed, while *dynamic* approaches insert the instrumentation code during run-time [Net04].

While instrumentation itself is versatile and can aid various tasks, such as profiling, debugging, and logging, it showed to be particular useful for the creation of dynamic bug finding tools, called sanitizers [SLR+19]. They typically use *compile-time instrumentation* to add additional code for detecting triggered bugs during program runtime. The most well-known sanitizers are probably AddressSanitizer [SBPV12], ThreadSanitizer [SI09], MemorySantizer [SS15b], and UndefinedBehaviourSanitizer [DLRA15], which have proven to be very efficient in uncovering vulnerabilities [Ser16].

However, access to the source code of the program under test is not necessarily given, and despite recent advances in static binary rewriting and instrumentation [WSB+17, KKC+17, DBXP20], dynamic binary instrumentation frameworks enjoy similar popularity as sanitizers. The most notable frameworks in this category are Pin [LCM+05], Valgrind [NS07], and DynamoRio [BA04].

### 2.3.3 Record & Replay

One often overlooked, but invaluable, recent advance for reverse engineering, debugging, and dynamic analysis is record-and-replay. The basic idea is to record every *non-deterministic* input to the program under analysis. These inputs are then replayed during a later run of the program, allowing for a deterministic re-execution.

This approach is beneficial for several reasons, as it allows to decouple dynamic analysis from the actual execution of the program [CGC08]. This is a tremendous advantage, as dynamic analysis techniques such as dynamic taint analysis or program slicing come with a performance overhead, which can disturb the functionality of the program under analysis. Having the replay available, the dynamic analysis tasks can be easily carried out during a replayed run of the program. Naturally, this allows also for scaling and applying different analysis techniques in parallel. Likewise, having a record enables analysts to run analyses which were neither foreseen nor anticipated during the recorded run of the program.

Additionally, manual debugging profits likewise from record-and-replay, as it can enable *reverse execution* and, in turn, timeless debugging [Hot16]. By now, a variety of systems for record-and-replay have been proposed, and popular tools are for instance Mozilla's RR for recording the execution of Linux user space programs [OJF+17], or PANDA for the execution of full, emulated systems [DGHH+15].

It is important to distinguish record-and-replay and traditional execution and memory traces. While traces enable like records scalable post-mortem analysis of programs, the collection of traces comes with a significant cost: hardware-supported tracing requires specialized hardware, while software-based tracing solutions often introduce performance overhead and require huge amounts of disk space as literally everything must be logged. Additionally, unlike for record-and-replay, the full program state is not available during trace analysis, and the important bits for the analysis have to be reconstructed.

### 2.3.4 Fuzz Testing

The term fuzzing, or fuzz testing, was coined by Miller et al. [MFS90] and describes an automated program testing techniques, whereby originally random input data is sent to the program under test. Nowadays, more sophisticated, "smart" fuzzing techniques than the original random fuzzing exists, and current fuzzing techniques can be classified based on the available information of the internal state of the program under test, the fundamental way new inputs are generated, and whether feedback from previous test runs are in this process [CCM+18].

One typically distinguishes between black-box fuzzing, in which the internal state of the program remains unknown to the fuzzer, white-box fuzzing, where all information are available, and grey-box fuzzing, where only limited program runtime information, such as code coverage, are exposed to the fuzzer.

Input generation can be roughly divided in mutation-based and generation-based generation. Mutation-based fuzzing starts from a given input and alters this input without information about any input specification, e.g., by flipping single bits.

In contrast, in generation-based fuzzing the input is generated according to prior known specification, which obviously yields higher coverage.

Feedback based, or guided-fuzzing, makes use of runtime information collected during the execution of the program, and is hence only used in combination with white-box or grey-box fuzzing. The gathered information is then used to improve the input generation for the next run, effectively *guiding* the fuzzer toward interesting points of the program. One especially often used information is the achieved code coverage: so called coverage-guided fuzzers evaluate generated inputs based on how much coverage they achieved and optimize for generating inputs uncovering new paths.

While fuzzing as testing technique is popular and is the active focus of many studies [MHH+18], a couple of fuzzers are particularly widely deployed in practice. Three of the most notorious examples are probably American Fuzzy Lop (AFL) [Zal14], libFuzzer [Ser15], and Peach [Pea17]. While the first two fuzzers are open source and allow for grey-box, coverage-guided, mutation-based fuzzing, the latter one represents a commercial solution for generation-based black-box fuzzing.

### 2.3.5 Symbolic & Concolic Execution

Symbolic execution, first proposed by King [Kin76], aims to identify what kind of inputs are required to reach a certain point in a program. The core idea is to execute a program with symbolic inputs which represent arbitrary values, rather than concrete ones. This requires special symbolic execution engines, which explore program paths while gathering constraints over the symbolic inputs. These constraints can then be solved to generate an input which would follow the path of interest in a concrete run of the program.

While it is unclear whether symbolic execution falls in the realm of static or dynamic analysis [Kel11], its worth as program testing technique is indisputable. The symbolic execution engine KLEE found 56 bugs in a corpus of 452 programs on its initial release [CDE+08] and is continuously used in both academia and industry up to this day [CN19], and Microsoft's SAGE [GLM+08] uncovered one-third of the bugs found during the development of Windows 7 [GLM12].

Considering these results, it is not surprising that the development of symbolic execution engines and techniques for binary software has experienced a renaissance in the last decade and an analyst has to choose between several frameworks, such as angr [SWS+16], BAP [BJAS11], Manticore [MMH+19], Miasm [Des12], Triton [SS15a], or S2E [CKC11].[2]

[2]This list is by no means exhaustive and merely highlights a small selection of popular open source frameworks.

Nowadays, symbolic execution is not only impactful on its own, but also became a powerful tool for enhancing dynamic testing techniques. A prominent example is *concolic execution* [GKS05, SMA05], in which a program is executed natively with a concrete input while an execution trace is collected. This trace is then re-executed inside a symbolic execution engine with symbolized inputs instead concrete ones. Now, new concrete inputs can be generated by analyzing the path constraints after every branch, and solving towards the untaken path by flipping the corresponding constraint. This process is usually executed in an automated loop, so that newly generated inputs are then again executed concretely and the newly collected traces are once more symbolically analyzed to generate more inputs, always aiming to cover additional paths in the program.

Furthermore, even fuzz testing, introduced in the last section, can benefit from symbolic execution. Tools like Driller [SGS+16] or QSYM [YLX+18] build upon the idea of interleaving symbolic execution with fuzz testing [MS07, Pak12]: whenever the fuzzer is "stuck", i.e., it does not reach new paths in the program, an input is analyzed concolically to generate a new input usable for the fuzzer.

All in all, promising testing approaches are enabled by symbolic execution, which is an interesting field of research with unsolved challenges on its own. However, so far symbolic execution has been mostly used to analyse desktop software, and first studies reported encouraging results when applying it to firmware analysis [DMRJ13, ZBFB14, SWH+15, HFT+17, CCF18].

# Chapter 3

# Understanding the Challenges of Dynamic Firmware Analysis

Adapting dynamic binary analysis and testing techniques to embedded devices is generally attributed to be challenging. In this chapter, we will first point out the specific challenges which have to be overcome to enable tractable dynamic firmware analysis. Then, we develop a security-focused classification for embedded systems which we will use through the remainder of this thesis. Furthermore, we demonstrate the effects of missing fault detection mechanisms on embedded devices. Finally, we will highlight and discuss different strategies to facilitate dynamic binary analysis for firmware.

## 3.1 The Challenges

Studies frequently point out domain specific challenges for dynamic binary firmware analysis in an ad hoc fashion. Examples for this include, for instance, studies on automated large-scale firmware analysis [CZF16, CWBE16], fuzz testing firmware [ZDY+19, MSK+18], or symbolic execution of firmware [DMRJ13, CCF18]. In this section, we structure the challenges encountered for dynamic firmware analysis in a set of five main challenges, which we will introduce one by one.

More precisely, these five challenges are: (1) firmware retrieval, (2) platform variety, (3) fault detection, (4) scalability, and (5) instrumentation.

### 3.1.1 Firmware Retrieval

Retrieving a device's firmware poses quite often a significant challenge. Sometimes, the firmware can be extracted directly from the device, but this requires access to the actual hardware. Even if the hardware is available, there is no guarantee that debug access allowing for easy firmware extraction is provided. In those cases a significant amount of time is often spent to obtain a foothold on the embedded system and more sophisticated, and invasive methods, such as flash dumping or bus snooping may be required [VOC18].

Another way to retrieve a device's firmware is via software update packages provided by the vendor. However, those are often packed in proprietary formats, some vendors even apply encryption on the update files [CZFB14]. Likewise, quite often firmware updates only update *parts* of the device's software, and, thus, obtaining a full image of the firmware remains challenging [CWBE16]. In fact, most large-scale studies on firmware analysis retrieve firmware via update packages, but fail to unpack a substantial amount of the collected dataset [CZFB14, CWBE16, FZX+16, TGC17, LFW+18, DPY18].

All in all, it is apparent that methods easing the process of extracting a firmware from the device or improving the efficacy of unpacking are needed for laying the way for the future of firmware analysis.

### 3.1.2 Platform Variety

As already discussed in Section 2.1, the definition of embedded systems spans a huge variety of devices. Not surprisingly, this leads to a very diverse landscape of hard- and software platforms with different processors, peripherals and operating systems.

In comparison to desktop systems, where the x86 family is indisputably the most adopted ISA, the embedded market is characterized by its diversity. While it is difficult to establish precise statistics about the deployment of different ISAs, a recent market study implies that chips based on the ARM, MIPS, PIC, MSP430, or AVR families are popular choices when designing new embedded systems [Max17]. Every single one of this ISAs comes with its own quirks, specificities, and variants, and even among the popular ones, comprehensive tooling, as present for desktop systems, is not necessarily available.

Besides the profoundly different architectures themselves, the hardware configuration they are shipped with is likewise diverse. While off-chip peripherals are added by manufacturers of embedded devices on demand, chip manufacturers combine CPU cores with the respective on-chip peripherals, and the variety of combina-

tions incredibly wide. For instance, there are more than 2500 different ARM-based chips registered by Keil, a company providing additional tooling for embedded developers [Kei19]. While all these chips belong to the same ISA family, the used peripherals and their implementations vary widely among those chips, leading to a massive heterogenity of hardware platforms.

Furthermore, depending on the type of the device, its processing power, and available peripherals, the granularity in which hardware abstractions are made is differing widely. E.g., powerful devices, can afford to run a full Linux-based operating system, which provides clean abstractions for the applications and programs. On the other end of the spectrum, low-power devices rarely run any operating system at all, and accesses to the hardware are directly inlined in application logic.

Hence, dynamic analysis tools and techniques for firmware need to be able to cope with the huge variety of different architectures, hardware platforms, and hardware abstraction levels.

### 3.1.3 Fault Detection

The majority of dynamic analysis techniques for finding vulnerabilities, such as fuzzing or concolic execution, rely on *observable crashes* as immediate consequences of faults occurring during a program's execution [TDMK18]. Desktop systems offer a variety of protection measures which are triggering a crash upon a fault, and while some of them are designed with security in mind (e.g., stack canaries), others are inherent architecture artifacts, such as segmentation faults caused by a Memory Management Unit. Unfortunately, these techniques are rarely present (or are very limited) on embedded devices.

Moreover, even when mechanisms leading to observable crashes are in place, monitoring those crashes can be complicated. In fact, while for desktop systems crashes are often accompanied by error messages, embedded systems may lack equivalent I/O functionalities.

As a result, dynamic firmware analysis and testing techniques cannot rely on the occurrence of observable crashes, and therefore need to deploy additional means to catch *silent corruptions*.

### 3.1.4 Scalability

Dynamic security testing for software is usually slow, as the speed of testing is often bounded by the execution speed of the program under test itself. Hence, testing is often parallelized for software targeting desktop systems, as multiple instances of the same software can easily be started and analyzed in parallel, e.g., via multi-processing or virtualization. This parallelization poses a substantial chal-

lenge when dealing with embedded devices. Obtaining numerous copies of the same physical device is often infeasible due to limited resources (e.g., financial), and to infrastructure requirements such as space and power supply.

Additionally, most testing and analysis techniques require a clean state of the program for the next run. This is easy to achieve for desktop systems, for instance via virtual machine snapshots or simple restarts of the program under analysis. Unfortunately, re-establishing a clean state on an embedded system can take a considerable amount of time (up to few minutes), as it often requires a full reboot of the device.

### 3.1.5 Instrumentation

As discussed in Section 2.3.2, being able to instrument code is an important asset for dynamic binary analysis.

Unfortunately, most binary rewriting tools which would allow static instrumentation are directly targeting the x86 ISA, which is rarely used in embedded systems. Although some specialized rewriting frameworks exist for ARM [KKC+17, HJO18], their application for enabling dynamic firmware analysis is somewhat limited. First, the firmware needs to be fully disassembled which can be difficult when it is provided as a raw binary [BBLD13]. Second, memory semantics, boundaries and data structures are lost in the compilation process and need to be recovered to add instrumentation. This requires a very challenging partial decompilation phase which goes far beyond binary rewriting. Third, as firmware resides most frequently in flash and ROM chips, injecting the instrumented code back to the embedded device can be a challenge on its own. Finally, the memory usage of embedded devices is often optimized to reduce costs, leaving little room for adding complex instrumentation.

Likewise, the tooling landscape for dynamic binary instrumentation is focused primarily on desktop systems. Although popular tools support by now more than just the x86 ISA [DCN+19], their operation is closely tied to to the abstractions provided by commodity operating systems. For instance, PIN [LCM+05] provides only instrumentation capabilities for Windows, Linux, and macOS software, while Valgrind [NS07] is actively only maintained for Linux, Solaris, Android, Darwin and illumos.

Hence, even if binary instrumentation is getting more and more adopted beyond configurations like x86/Linux, significant work is required before solutions become tractable for embedded systems.

## 3.2 Classification of Embedded Systems

As pointed out in Section 3.1.2, the diversity of used processors, peripherals, and hardware abstractions in embedded systems is enormous. Hence, we believe that a security-focused classification of embedded systems is needed, which we provide in this chapter.

For this purpose, let us first recap the generic definition of embedded devices we introduced in Chapter 2:

> *Embedded devices are separated from modern general-purpose computers by two common characteristics: a) embedded systems are designed to fulfill a special purpose, and b) embedded systems often interact with the physical world through a number of peripherals connected to sensors and actuators.*

These two criteria cover a wide variety of devices, ranging from hard disk controllers to smart thermostats, from digital cameras to PLCs. These families can be further classified according to several aspects, such as their actual computing power, their unit cost, the field of usage, the extent to which they interact with the environment, or the timing constraints imposed on the device.

Yet, these classifications tell very little about the type of security mechanisms that are available on a given device. Therefore, we propose in this thesis to classify embedded systems according to the type of operating system they use. While the operating system is certainly not the only source of security features, it is responsible for handling the recovery from faulty states, and it often serves as building block for additional, more complex, security primitives.

By using the operating system as basis for a classification scheme, we can divide embedded devices in three classes, which we present in the following. For better readability, we will use Type-I, Type-II, and Type-III indifferently with the corresponding class name in the remainder of this thesis when referring to embedded devices.

### 3.2.1 Type-I: General Purpose OS-based Devices

General purpose Operating Systems are often retrofitted to embedded systems, especially when complex logic is needed such as for routers, IP cameras, or access points.

However, in comparison to the traditional desktop or server counterparts, embedded systems typically follow more minimalistic approaches. For example, the

Linux OS kernel is widely used in the embedded world, where it is typically coupled with lightweight user space environments (e.g., `busybox` and `uClibc`). In those settings, interaction to custom hardware is most of the times carried out via special device drivers and the abstractions provided by well-known operating systems make this class a popular target for dynamic analysis.

### 3.2.2 Type-II: Embedded OS-based Devices

In recent years, custom operating systems for embedded devices have gained popularity. These systems are particularly suitable for devices with low computational power, and while advanced processor features such as a Memory Management Unit (MMU) may not be present, a logical separation between kernel and application code is still present. Operating systems such as `uClinux`, `ZephyrOS` or `VxWorks` are examples for these systems and they are usually adopted on single-purpose user electronics, such as LTE modems or DVD players.

### 3.2.3 Type-III: Devices Without an OS-Abstraction

These devices adopt a so called “monolithic firmware”, whose operation is typically based on a single control loop and interrupts triggered from the peripherals in order to handle events from the outer world. Monolithic approaches can be found in a large variety of controllers of hardware components, like CD reader, WiFi cards or GPS dongles. The code running on these devices can be completely custom, or it can be based on *operating system libraries* such as `Contiki`, `TinyOS` or `mbed OS 2`[1]. While those libraries are providing abstractions to programmers, the resulting firmware will contain both system and application code compiled and linked together, and, thus, forms a monolithic software.

### 3.2.4 Devices in this Thesis

Throughout this thesis, we analyze and use several devices, ranging from a home router to the I/O controller on a PLC. Naturally, the devices can be categorized according to the classification presented in this chapter, as shown in Table 3.1.

As Type-I device, we chose the Linksys EA6300v1. This router uses an ARM Cortex-A9 CPU core and runs an embedded Linux operating system together with uclibc and busybox, which represents a very common configuration. Additionally, the router exposes an USART interface on its PCB which provides a root shell to the Linux OS.

[1] Interestingly, and despite the name, mbed OS 2 (also referenced as "mbed OS classic") is an operating system library. Later versions, on the other hand, are actual embedded OS and would serve as building block for Type-II devices.

Furthermore, we use very diverse Type-II devices: An IP Camera, a digital single-lens reflex (DSLR) camera, and a solid state disk. The IP camera, a Foscam FI8918W, hosts a uClinux-based firmware on a Winbond W90N745 SoC based on an ARM7TDMI core. The PCB of the camera exposes a pinout for a USART interface and JTAG for the core is enabled, which eases analysis and debugging.

The Canon EOS 60D DSLR camera, on the other hand, uses a DIGIC 4 chip, a SoC specifically developed by Canon for digital cameras. This SoC is built around an ARM core and runs DRYOS, a proprietary real-time operating system for digital cameras and camcorders. We chose this camera because a good emulator for its firmware is readily available as part of the *Magic Lantern* project[2].

Our third Type-II device is the Crucial MX 100 SSD by Micron Technology. This SSD is based on a dual-core ARM SoC (Marvell 88SS9189) and an MSP430 MCU. This disk is particularly interesting as it has been analysed in previous studies and has JTAG and USART interfaces exposed on its PCB [CRB17, MvG19].

As real world Type-III device we use the I/O controller embedded on the Allen Bradley CompactLogix 5370 PLC. The controller is based on a Texas Instrument Stellaris LM3S2793 microcontroller which revolves around an ARM Cortex-M3 core. Like the SSD, this particular controller exposes its JTAG interface on the PCB and has been subject to prior research [GBC+17].

Additionally, we use a STMicroelectronics Nucleo L152RE development board as Type-III test platform at various occasions throughout this thesis. This board is built around a Cortex M3 core and integrates a ST-LINK debugger, providing easy access to the core's JTAG interface via USB. Using a development board has the benefit of a readily available toolchain and a well-known hardware platform. This allows us to concentrate our focus on the development of dynamic binary analysis techniques for firmware and removes the otherwise required reverse engineering of hardware and firmware.

| Platform | Vendor | Model | Device Type | Occurrences |
|---|---|---|---|---|
| Router | Linksys | EA6300v1 | I | Section 3.3 |
| IP Camera | Foscam | FI8918W | II | Section 3.3, 6.4 |
| Digital Camera | Canon | EOS 60D | II | Section 6.4 |
| Solid State Disk | Micron | Crucial MX100 | II | Section 6.4 |
| Development Board | STMicroelectronics | Nucleo L152RE | III | Section 3.3, 5.2, 5.4 |
| PLC I/O Controller | Allen Bradley | CompactLogix 5370 | III | Section 5.1 |

**Table 3.1:** Devices used and analyzed in this thesis.

[2] https://magiclantern.fm/. The project aims to add additional features to Canon EOS cameras and uses QEMU for testing.

## 3.3 Investigating the Lack of Fault Detection

Challenge-3 states that embedded devices often lack even simple fault detection mechanisms which are commonly found on desktop systems. This is especially daunting for dynamic security testing, as most tools and techniques implicitly assume that triggered bugs would lead to *observable crashes*. Unfortunately, many bugs which would lead to crashes on desktop systems result into *silent corruptions* on embedded systems due to the lack of fault detection mechanisms.

However, in comparison to the other challenges presented earlier in this chapter, which have either been the subject of previous studies (e.g., firmware retrieval [VOC18]), or are well understood (e.g., scalability), the implications of lacking fault protection require further investigation. In this section, we will showcase our experiments which demonstrate the severity of missing fault protection on the example of memory corruptions across different classes of devices.

### 3.3.1 Experimental Setup

In order to study how memory corruptions behave across different computing systems with different levels of fault detection mechanisms, we conducted a number of experiments. The goal of these experiments is to trigger the same memory corruption conditions on different systems to analyze whether they yield to observable crashes or result in silent corruptions.

We selected one device for each device class presented in Section 3.2, and compare the results with a baseline system consisting of full-fledged GNU-Linux desktop OS. All systems are ARM-based and we recompiled each firmware image to obtain comparable results. We introduced artificial vulnerabilities in two popular and widely used libraries: *mbed TLS*, an SSL library designed for both embedded and desktop system, and *expat*, a popular XML parser. The two are good candidates for our experiments because memory corruptions vulnerabilities were previously found in both of them, and because they are popular, open source, and present in many modern embedded devices.

To analyze their behavior in a realistic context, we chose existing Commercial Off-The-Shelf (COTS) products: a router to represent Type-I systems, and an IP camera for Type-II devices. We compiled our vulnerable application for those targets and then loaded it on the device. For the monolithic class, however, obtaining a COTS device with customizable firmware is difficult, as their firmware usually consists of only one custom binary blob responsible for the entire operation of the device. Therefore, we used a development board with publicly available software for peripheral interaction. For this device we included our test code in the firmware, compiled it and then loaded the firmware on the device. Table 3.2 shows a

summary of our three test platforms, including information about which C library is used and whether a MMU is present. Besides the operating system, these properties mainly determine the behavior of a system in case of a memory corruption.

| | Platform | Manufacturer & Model | Operating System | LIBC | MMU |
|---|---|---|---|---|---|
| **Desktop** | Single Board Computer | Beaglebone Black | GNU/Linux | glibc | ✓ |
| **Type-I** | Router | Linksys EA6300v1 | Embedded Linux | uclibc | ✓ |
| **Type-II** | IP camera | Foscam FI8918W | uCLinux | uclibc | ✗ |
| **Type-III** | Development Board | STM Nucleo-L152RE | None | libmbed | ✗[3] |

**Table 3.2:** Devices selected for the experiments.

### 3.3.2 Artificial Vulnerabilities

Since our focus is not the discovery of new bugs but rather the analysis of the effects of memory corruptions on embedded systems, we inserted several vulnerabilities leading to memory corruptions in our test samples. Specifically, we used stack-based buffer overflows and heap-based buffer overflows as examples of spatial memory corruptions, and null pointer dereferences and double free vulnerabilities as examples of temporal memory corruptions. Additionally, we also inserted a format string vulnerability that can either be used for information leakage or for arbitrary memory corruptions.

Our approach of bug insertion was inspired by the one used in LAVA [DGHK+16], i.e., we ensured that each vulnerability had its own independent trigger condition. However, as we only had to inject a limited number of vulnerabilities, we *manually* selected the vulnerable paths and the position of each bug. Likewise, for the purpose of our experiments we did not need "realistic" checks or path conditions particularly difficult to explore. Therefore, we simply added custom branches in the code that triggered the various vulnerabilities based on the length of the user-provided payload. Listing 3.1 shows four of the inserted vulnerabilities inside the expat library.

[3] Note that the microcontroller used by this device can be equipped with an optional Memory Protection Unit (MPU), which provides basic memory protections. While present on this specific Type-III device, we do not utilize its features, as this is a very common scenario and the most problematic case.

**Listing 3.1:** Examples of artificial vulnerabilities.

```
XML_Parse(XML_Parser parser, const char *s, int len, int isFinal)
{
  char overflowable[128];
  [...]
  //this returns a heap object
  void *buff = XML_GetBuffer(parser, len);
  [...]
  //trigger immediate stack-based buffer overflow
  if (len == 1222){
    memcpy(overflowable, s, len);
    return;
  }
  //this will cause a null pointer dereference
  else if (len == 1223){
    buff = NULL;
  }
  //causing a heap-based buffer overflow
  if (len == 1225){
      memcpy(buff, s, len);
      memcpy(buff + 1225, s, len);
  }
  else{
    memcpy(buff, s, len);
  }
  //cause an uncontrolled format-string vulnerability
  if (len == 1224){
    printf(buff);
  }

  [...]
}
```

### 3.3.3 Observed Behavior

The goal of this experiment was to observe the behavior of each device after sending malicious inputs that trigger one of the inserted vulnerabilities. As expected, the software running on GNU/Linux desktop crashed each time it was provided with a malicious input that triggered the vulnerability.

However, the embedded devices were not always able to detect the fault and, in some cases, they even continued the execution with no visible effects – despite the fact that the underlying memory of the system was corrupted. To better differentiate between the different behaviors we observed in our experiments, we categorize our observations in six possible results:

[R1] **Observable Crash (✓)** – The execution of the device under test stops and a message or other visible effect is easily observable. In less optimal cases, no detailed information about the causes of the crash is produced (we mark these cases as `opaque` in Table 3.3).

[R2] **Reboot (✓)** – The device immediately reboots. For Type-III devices there is no difference between a crash and a reboot because they are monolithic applications. However, for Type-I and Type-II devices a given service (e.g., a web server) can crash while the rest of the embedded system may still continue to work properly.

[R3] **Hang (!)** – The target hangs and stops responding to new requests, possibly stuck in an infinite loop.

[R4] **Late Crash (!)** – The target system continues its execution for a non-negligible amount of time and crashes afterwards (e.g., when the connection is terminated).

[R5] **Malfunctioning (✗)** – The process continues, but reports wrong data and incorrect results.

[R6] **No Effect (✗)** – Despite the corrupted memory, the target continues with no observable side-effects while being in a faulty state.

Immediately observable crashes and reboots (R1 and R2) are the preferred outcomes of an experiment as during a fuzzing session they allow to immediately identify the responsible input.

Hangs and late crashes (R3 and R4) can be more difficult to deal with, in particular when the crash is delayed long enough that a fuzzer may have already sent multiple other inputs to the target system and the input responsible for the corruption will therefore be difficult to identify. However, the presence of a fault is still observable in these cases. Things get even more complex when a device starts malfunctioning (R5). In this case, there are no crashes at all, but the results provided to certain requests may be incorrect. To detect this case, a fuzzer would need to know what is the correct output for each input it sends to the system – something which is very seldom the case in security testing. A possible workaround can consist in inserting between two consecutive inputs a number of functional test requests for which the output is known. However, even when this solution is sufficient to detect the malfunction, it introduces a considerable delay in the fuzzing experiment. Finally, the worst case is when the device continues its operation with no observable side effect (R6). In fact, since part of the device memory has been corrupted, there may be side effects or unexpected behaviors in the future.

Table 3.3 shows the result of our experiments. It is clear that the fewer features are provided by an embedded platform, the less likely the system is to detect memory corruptions. An interesting observation is already the difference between a full

scale GNU/Linux and an embedded Linux for heap-based buffer overflows and double free corruptions. While on the desktop system the inlined heap consistency checks provided by the standard C library are triggering a verbose crash quickly after the corruption, the embedded Linux continues and crashes at a later point in time, often just during the `exit()` handler.

The table also shows that corrupting inputs for Type-II and Type-III devices are very rarely triggering a crash. This provides a perfect example of how common are *silent* memory corruptions in real-world embedded systems.

Another important observation can be made when looking at the devices' behavior for the format string vulnerability. Both the embedded and the full scale GNU/Linux are reporting segmentation faults, due to the attempt to access unmapped memory. However, `uClinux` and the monolithic device are continuing execution, which is a result of the lack of an MMU. This shows that the MMU plays a very important role when it comes to detecting memory corruptions.

A similar behavior can be observed on the results of the null pointer dereference test. The processes running on both the GNU/Linux desktop and on the embedded Linux are correctly crashing once the program tries to write memory to the NULL-address range. With the same vulnerability, the monolithic device will continue execution although data has been written to the exact same address. In fact, as the execution of the firmware is not dependent on the content of the memory at this location, this memory corruption does not influence the behavior of the system.

The result of the same test on the `uClinux` system is particularly interesting: after the NULL write, the device hangs for few seconds and then reboots. This is not surprising as in `uClinux` the kernel is mapped in the lower part of the memory (so writing at address `0x0` corrupts kernel memory). The reboot, on the other hand, is possibly caused due to an hardware watchdog that detects the hang and automatically restarts the device as a recovery mechanism.

To summarize, while in certain conditions silent memory corruptions can occur also in traditional desktop environments, our experiments show that they are often the rule and not the exception in the embedded world. As popular dynamic testing techniques such as fuzzing rely on observable crashes to detect bugs, the limited support for fault detection can have very severe consequences for the effectiveness of security testing for embedded devices.

| Platform | Desktop | Type-I | Type-II | Type-III |
|---|---|---|---|---|
| Format String | ✓ | ✓ | ✗ | ✗ |
| Stack-based buffer overflow | ✓ | ✓ | ✓ (opaque) | ! (hang) |
| Heap-based buffer overflow | ✓ | ! (late crash) | ✗ | ✗ |
| Double Free | ✓ | ✓ | ✗ | ✗ (malfunc.) |
| Null Pointer Dereference | ✓ | ✓ | ✓ (reboot) | ✗ (malfunc.) |

**Table 3.3.A** Expat.

| Platform | Desktop | Type-I | Type-II | Type-III |
|---|---|---|---|---|
| Format String | ✓ | ✓ | ✗ (malfunc.) | ! (hang) |
| Stack-based buffer overflow | ✓ | ✓ | ✓ (opaque) | ! (hang) |
| Heap-based buffer overflow | ✓ | ! (late crash) | ✗ | ✗ |
| Double Free | ✓ | ! (late crash) | ✗ | ✗ |
| Null Pointer Dereference | ✓ | ✓ | ✓ (reboot) | ✗ |

**Table 3.3.B** mbed TLS.

**Table 3.3:** Observed system behaviour for triggered memory corruptions.

## 3.4 The Paths to Binary Firmware Analysis

In the previous sections, we have shown that embedded systems come in all forms and with very different characteristics. While testing a Type-I device may be very similar to testing a desktop system, Type-III devices are more difficult to analyze, especially due to missing fault detection mechanisms: memory corruptions on Type-III systems rarely result in an immediate crash, imposing a significant challenge to automatically identify when a vulnerability has been triggered.

So how can we develop efficient dynamic binary analysis techniques to firmware when no reliable feedback is available? Again, the vast diversity of existing devices makes it difficult to find a general answer to this question. Therefore, in this section we present six different options that may be available to the tester, and we discuss both advantages and limitations of each solution.

### 3.4.1 Physical Re-Hosting

In certain cases, the analyst may be able to move the binary code to a different target device, for example to relocate a process from a Type-II device to a more test-friendly Type-I device or rehost a full program from a Type-I device on a regular Linux desktop system. This may also improve *scalability*, if the new device is cheaper and more readily available than the original one or if the new target is a regular computer.

However, on top of the difficulties of transferring the program to a different system, methods relying on this approach have another major drawback. In fact, it may be difficult to reproduce bugs found on the new target system in the original device (where they may even not be present at all due to the different architecture or due to changes introduced by moving the binary) and conversely, bugs that are present on the original target may not be present on the new target.

### 3.4.2 Full Emulation

Especially for Type-I devices, images of the device firmware are often available to the analyst, either because they are extracted directly from the device or because they are obtained from a firmware update package available from the manufacturer. Costin et al. [CZF16] and Chen et al. [CWBE16] have shown that, under certain conditions, applications extracted from Type-I firmwares can be *virtually rehosted*, i.e., they can be executed inside a generic operating system running on a default emulator. Likewise, the Qemu STM32 [Bec13] project, which extends Qemu to emulate the STM32 chip, shows that when complete hardware documentation is available, with a considerable effort to implement the hardware emulation it is also possible to fully emulate Type-III firmware images.

This solution can greatly improve dynamic testing. First of all, tests can be conducted *without* the presence of the physical device, thus allowing for a much greater parallelization. Second, dynamic instrumentation techniques can be easily applied and the emulator can be used to collect a large amount of information about the running firmware. The disadvantage of this solution is that it is only applicable when all peripherals being accessed by the target are known and can be successfully emulated, which is unfortunately rarely the case.

### 3.4.3 Partial Emulation

If full emulation remains impractical in most circumstances, partial emulation, also called hardware-in-the-loop emulation, can still provide benefit while conducting dynamic testing.

This approach was first proposed by Avatar [ZBFB14] and Surrogates [KKM15] for Type-III devices and then extended to Type-I systems in PROSPECT [KBK16, KPK14] and Charm [TTZ+18]. The general idea behind these solutions is to use an emulator (in which the firmware code is executed) modified to forward peripheral interactions to the actual physical device. The result provides the advantages of a full emulation solution without the burden of knowing and emulating I/O operations. However, what this solution gains in flexibility is sacrificed in performance (due to the additional interaction with the real device) and scalability (due to the current need of pairing each emulated instance with a physical device).

### 3.4.4 Symbolic Execution

Another approach to deal with platform variety in general, and the difficulty of implementing peripherals in particular, is to leverage symbolic execution. The idea is to execute the firmware inside a symbolic execution engine and consider the results of hardware interactions as symbolic data.

This approach has been shown for instance by Firmalice [SWH+15] and FirmUSB [HFT+17]. While this approach easily allows for scalability and additional instrumentation inside the symbolic execution engine, it inherits all the limitations of symbolic execution. Even worse, some of the limitations are amplified when applying symbolic execution to embedded systems. Consider for instance the widely acknowledged problem of path explosion [BCD+]. Due to the highly asynchronous environment presented by some systems, interrupts can potentially introduce a new state *at any time* and symbolic execution engines need to be modified to integrate specialized interrupt scheduling mechanisms [DMRJ13, HFT+17]. Likewise, symbolic execution typically benefits from *environment modeling*, which uses common abstractions provided by system libraries and operating systems which may not be present on Type-II and Type-III devices.

### 3.4.5 Software-based Instrumentation

Injection of code, both statically and dynamically, has been successfully carried out on powerful embedded systems such as routers [CS11], PLCs [GBC+17], and printers [CCS13] in the past. While these are promising results and show the feasibility of instrumentation for enabling dynamic binary firmware analysis, it does not come without problems.

As pointed out in Section 3.1.5, embedded systems, especially on the low-cost end, are frequently size-constrained, and may not have enough space for statically instrumented binaries. Likewise, the deployment of binary instrumentation tools can easily be hindered by those very same space constraints, to less computing power, or missing abstractions provided by an operating system.

Hence, the adaption of popular instrumentation techniques and tools may be feasible for Type-I and Type-II, but remains challenging for Type-III systems.

### 3.4.6 Hardware-Supported Instrumentation

If the tester has access to a physical device with advanced hardware instrumentation mechanisms (such as real time tracing), it may be possible to collect enough information during the execution of the device for retrofitting dynamic analysis techniques. For instance, chip manufacturers often embed hardware tracing features such as ARM's Embedded Trace Macrocell (ETM) and Coresight Debug and Trace, or Intel's Processor Trace (PT) technologies [Jam13] [4]. ARM tracing mechanisms are optional components of processors ("IPs"), which may be optionally included in the processor. Multiple variants tracing exist, such as tracing only branches, all instructions, or also all memory accesses – and different techniques rely on different debug ports (e.g., dedicated trace ports, *Single Wire Debug* (SWD) or *Single Wire Output* (SWO) ports).

Unfortunately, the availability of such tracing hardware is variable. In lower-end devices (typically Type-III devices), manufacturers tend not to include any tracing capabilities, because of the relatively large impact on the chip surface, and therefore on the cost, that such mechanisms would incur. Development devices may have such facilities (sometimes when the micro-controller design is tested on FPGA before manufacturing) but this is less frequent in commercial production devices. Finally, in some cases debug access may be present but not available to prevent third-party analysis.

For example, while looking for test devices to conduct our experiments we encountered devices where the tracing support was either deactivated for security

[4] However, only available on recent high performance processors.

reasons, or where the tracing pins where not routed on the circuit board (PCB), or multiplexed on pins which are used for another purpose.

In summary, when testing real world devices, the chances of finding an available and usable hardware tracing support are quite low.

### 3.4.7 Summary & Next Steps

To summarize, the six presented approaches for improving dynamic firmware analysis can be divided into two fundamental distinct paradigms: instrumentation and rehosting.

Instrumentation-based approaches either require special hardware features, or modification of the firmware running on the device. Unfortunately, as we already explained above, this can be challenging, especially when testing Type-III devices.

On the other hand, rehosting-based approaches, in which the firmware is transferred to, and executed in, a different execution environment than the original embedded system has been shown to be applicable across all three classes of devices. Additionally, execution of the firmware inside an emulator or symbolic execution engine allows for dynamic instrumentation, and hence, rehosting experiences the benefits of instrumentation-based approaches as well. Therefore, we will focus on rehosting-based approaches in the remainder of this thesis.

# Chapter 4

# Rehosting for Fun & Profit

As concluded in the last chapter, rehosting appears to be a viable approach for tackling the challenges of dynamic firmware analysis. In this chapter, we will first survey existing rehosting solutions and then make the case for multi-target orchestration. Afterwards, we will present `avatar`$^2$, our multi-target orchestration framework which provides flexible dynamic analysis primitives and is capable of rehosting embedded systems' firmware.

## 4.1 Rehosting: State of the Art

In this section, we will review the literature on the topic of *virtual rehosting*, which comprises all approaches for rehosting firmware, or parts of it, into a virtual execution environment. Hence, we will present various emulation, partial-emulation, and symbolic execution based rehosting systems, their contributions to dynamic firmware analysis, and their limiting factors. Additionally, we visualize and contextualize existing work in Figure 4.1 and provide insights about key aspects, such as the target device types, or whether source code is required.[1]

### 4.1.1 Full Emulation

Embedded device and OS vendors sometimes develop proprietary emulators for sale, in-house testing, or 3rd party development [Fit18]. However, even if readily available, device-specific emulators are typically ill-suited for security research, as they are typically closed-source and lack integration with popular dynamic analysis tools. Hence, most prior work focuses on rehosting the target system into a general-purpose, extendable emulator.

[1] Note that we merge Type-I and Type-II systems into a single category in this contextualization, as most rehosting solutions who support Type-I systems support Type-II systems and vice versa.

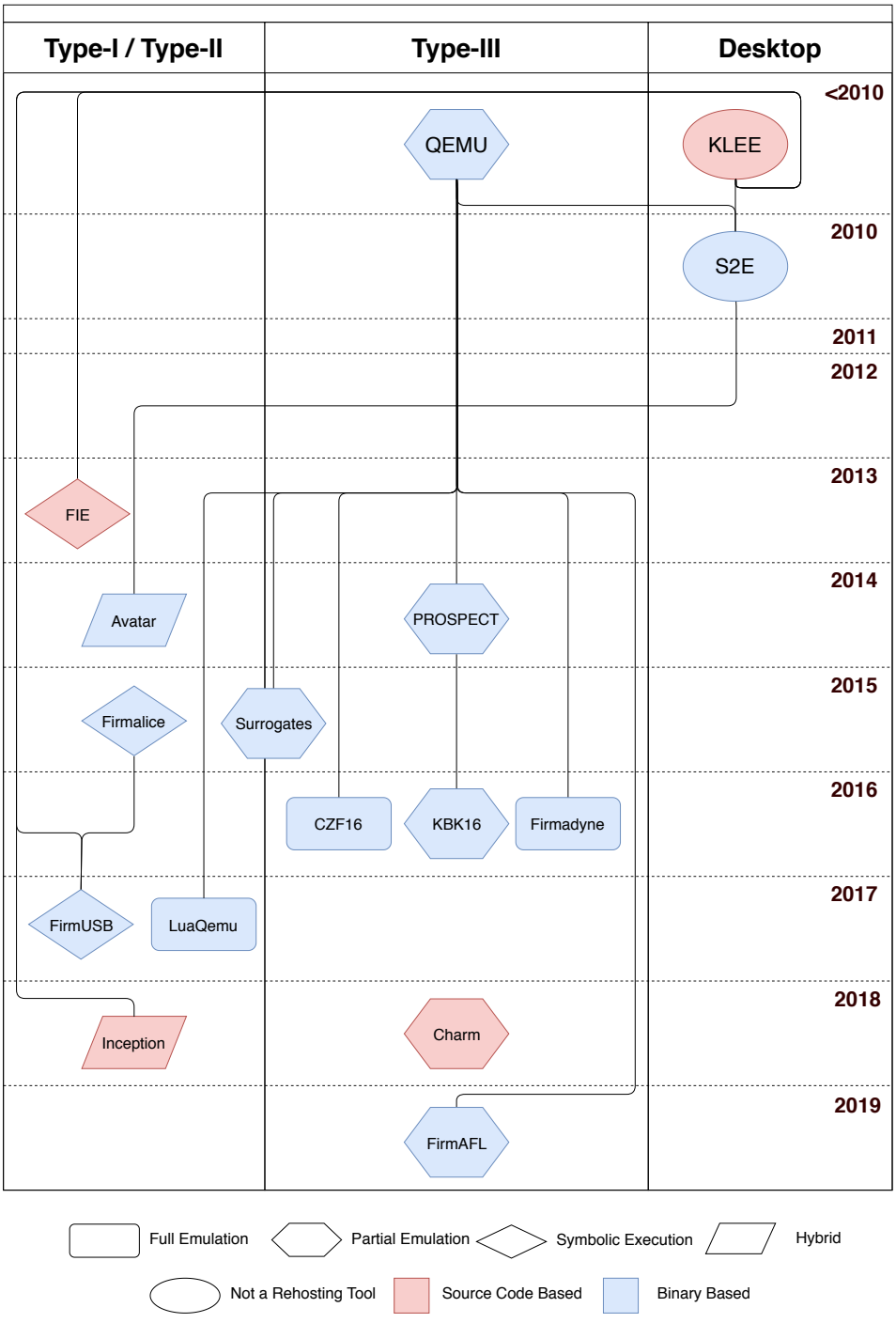


**Figure 4.1:** Timeline of rehosting.

QEMU [Bel05] is the most popular emulator for rehosting various hardware platforms. It supports a large variety of ISAs, and is free and open-source. Various forks exist to enable the emulation of specific hardware platforms (e.g., STM32 microcontrollers [Bec13], or ESP32 SoCs [Ebi16]), and the emulator serves frequently as base platform for more generic rehosting solutions.

For instance, **Firmadyne** [CWBE16] uses QEMU for dynamic large-scale emulation-only analysis of firmware images. **Firmadyne** executes Linux-based firmware by executing an instrumented Linux kernel alongside the original user space file system and binaries which have been extracted from the firmware image. This enables analysis of 23035 firmware images, scraped from vendor websites. While the majority of non-standard peripheral interactions are ignored, a custom user-space stub is used to replace a common key/value store in non-volatile memory (NVRAM). Of the 8617 firmware images selected for testing, 1971 booted successfully and connect to a network in Firmadyne. The authors then performed automated dynamic analysis of the rehosted firmwares by accessing web pages, collecting Simple Network Management Protocol (SNMP) information and testing whether the firmware is vulnerable to a specific set of known and hand-crafted exploits. Additionally, the framework implements features to aid manual analysis, such as dynamic tracing of executed code, and the injection of a special application for modifying the firmware image on-the-fly during emulation.

A similar approach to the one demonstrated by Firmadyne was described in [CZF16] with a focus on analyzing web interfaces. QEMU is used to run a generic Linux kernel for executing web servers embedded on the unpacked file systems of the target firmwares. The results of a preliminary static analysis phase are then used to drive a variety of diverse dynamic testing tools to uncover previously unknown vulnerabilities.

Another recent approach is presented by FirmAFL [ZDY+19]. In this approach, the authors combine a mix of full-system and user-mode only emulation provided by QEMU to enable high-throughput fuzz-testing of embedded Linux applications with AFL.

While all three of these systems purely focus on Linux-based firmware and are not able to perform analysis on firmware for Type-II or Type-III systems, a different approach is presented by **LuaQemu** [Ral17]. In this approach, QEMU is adjusted to emulate a on-the-fly configurable hardware platform to embed a Lua interpreter, which allows for providing custom hooks for accesses to not implemented hardware peripherals. LuaQemu has been used to allow manual dynamic analysis of a Broadcom WiFi SoC embedded on a Samsung Galaxy S6.

### 4.1.2 Partial Emulation

Challenge-2, platform variety, and the resulting diversity of peripherals is a core challenge for providing emulators. Hence, various partial emulation approaches forward hardware accesses to the physical embedded system to avoid tedious case-by-case implementation of peripherals.

**Prospect** [KPK14] for instance uses the fact that peripheral interaction on Linux is frequently carried out via device drivers exposing character devices. The system uses QEMU to rehost Linux firmware with a customized kernel and forwards accesses to the character devices to the physical hardware via user-space stubs. The authors evaluated PROSPECT by performing a security audit for a proprietary fire alarm system, which included fuzz testing and manual dynamic analysis of the firmware. In a follow-on effort to Prospect, Kammerstetter et al. [KBK16] improve the system's scalability by caching expected peripheral behavior and resetting the cache whenever a peripheral's behavior diverges from what was expected.

A similar approach is taken by **Surrogates** [KKM15], which uses the JTAG debug interface of an embedded device to forward the raw memory accesses to—and interrupts from—peripherals without implementation in QEMU. To achieve near-realtime access speeds, the system uses an FPGA on a PCIe card to drive the JTAG interface. Due to the forwarding of raw interaction in near-realtime, Surrogates is capable of rehosting arbitrary firmware and is not restricted to Linux-based devices. While the system is not directly used for dynamic analysis purposes, the near-realtime execution speed allows for powerful analysis and testing techniques, such as fuzzing, concolic execution, or taint tracking.

A more recent system, **Charm** [TTZ+18], forwards peripheral interactions to a real phone for fuzz testing Android device drivers. It uses hardware-based virtualization, the hardware's device tree and manual modified device drivers to forward raw interaction via USB 3.0. While this allows low-latency forwarding suitable for a fuzzing campaign, Charm's rehosting and analysis capabilities are limited to peripherals with open source drivers for android.

### 4.1.3 Symbolic Abstractions

Another solution to deal with the variety of hardware platforms with unknown peripheral behavior is symbolic abstraction. This strategy may result in lower fidelity than partial emulation due to the introduction of unrealistic behavior, but can help exercising additional code paths for bug-finding purposes.

One example of this approach is **FIE** [DMRJ13], which executes MSP430 firmware inside the KLEE symbolic execution engine. Additionally, the analyst has to

specify a *Memory Spec* and an *Interrupt Spec* to deal with memory locations used by peripherals and interrupts generated by them. As a result, reads and writes to peripherals with unknown behaviour can be treated symbolically, and the locations where the execution flow can be changed by interrupts are marked for spawning new states in the symbolic execution engine. FIE was used to enable symbolic exploration and white-box fuzzing. Due to fundamental limitations with symbolic execution (specifically, state explosion), FIE could only work on simple programs running without a complex operating system. Additionally, as the system is based on KLEE, it requires access to the firmware's source, further constraining its usability. Nevertheless, on these simple programs, the authors were able to use FIE for symbolic exploration and white-box fuzzing.

**Firmalice** [SWH+15] uses symbolic execution to identify authentication bypasses (e.g., backdoors) in binary firmware. It uses static analysis to identify the portions of a firmware's logic related to authentication, which are then sliced and symbolically executed with angr [SWS+16]. To avoid direct interaction with hardware, the tool uses symbolic function summaries and treats memory accessed in interrupt handlers as symbolic.

Another approach to leverage symbolic execution for binary firmware analysis is presented by **FirmUSB** [HFT+17], which uses domain-specific knowledge of the USB protocol to create symbolic abstractions for analyzing firmware of USB devices. As backend for symbolic execution, FirmUSB supports both FIE and angr, while lifting the firmware code to the Immediate Representations (IRs) used by those engines. Although the tool can only analyse USB firmware based on the 8051 architecture, the mix-in of domain-specific knowledge to ease analysis is comparable to using operating system abstractions like Firmadyne or Prospect.

### 4.1.4 Hybrid Approaches

Some rehosting systems combine partial emulation with symbolic abstractions to bound the complexity of the symbolic analysis. **Avatar** [ZBFB14], for instance, uses S2E [CKC11] for emulation, selective symbolic execution, and dynamic analysis, while forwarding accesses to hardware via JTAG or a custom GDB stub. Additionally, the framework allows to transfer the state between hardware and emulator for natively executing hard to emulate portions of the firmware. While Avatar's approach to rehosting requires no modifications to the running software, it requires non-negligible effort per-device to set up introspection and cannot be used at scale. Additionally, the latency introduced by peripheral forwarding is significant, and regular interrupts can easily exhaust the bandwidth for forwarding hardware interactions. Avatar was initially evaluated by dynamically analyzing a hard disk bootloader, a wireless sensor node, and a GSM feature phone. Although

the partial emulation strategy is similar to Surrogates, the lack of specialized hardware to achieve near-realtime execution speeds limits the analysis capabilities of Avatar to Type-II and Type-III firmware without timing constraints.

Another approach for hybrid full-system rehosting is provided by the **Inception framework** [CCF18]. The framework consists of a custom JTAG debugger for near real-time forwarding, a symbolic execution engine based on KLEE, and a translator for merging lifted and compiled LLVM bitcode to cope with inline assembly within the firmware's source code. Although Inception enables full-system testing, its implementation is tied to ARM Cortex-M3 chips and requires the source code of the firmware, constraining its usability.

## 4.2 The Case for Multi-Target Orchestration

In the last section, we briefly surveyed the landscape of rehosting systems for dynamic security testing of embedded systems. A key insight of this analysis is that most of those systems are based on either KLEE [CDE+08], QEMU [Bel05], or S2E [CKC11]. However, the majority of binary analysis techniques are not implemented in these frameworks, but rather in independent tools to demonstrate their effectiveness. While some tools are simple prototypes, others are more mature and have earned significant popularity over the years.

Unfortunately, most of these techniques' implementations are deeply coupled with their dynamic analysis frameworks and are not easy to integrate into other tools. As every framework aims to obtain the best analysis possible, re-implementations of techniques which are part of other tools are quite common [BJAS11, SWS+16].

While each framework has its own strengths and weaknesses, they all share a property: the analysis state is tightly coupled to the specific framework. This is due to a variety of reasons, including incompatible design choices (such as the use of an *intermediate representation* specific to a certain tool or the abstract representation of the program state in a custom format) or the fact that developers often implement tools as standalone systems that are implemented to be flexible to use but are nevertheless difficult to integrate with other solutions. Besides leading to duplication of work, as different tools are often implementing the same dynamic binary analysis techniques independently of each other, this prevents the full potential of binary analysis from being unleashed.

So far, little effort has been invested towards a better interaction between different frameworks, and not only in terms of re-using analysis results, but also by sharing the internal analysis state to external components. This prevents analysts from being able to combine different tools to exploit their strengths and tackle complex problems which requires a combination of sophisticated techniques.

We believe that a framework which is able to facilitate the interoperability among multiple binary analysis tools would ease the development of rehosting platforms, as it would let the analyst choose multiple execution environments, which are later used to perform their tasks on the *same* execution state.

## 4.3 The Avatar$^2$ Framework

In this section, we will present `avatar`$^2$, the framework we developed to flexibly interconnect multiple binary analysis tools, such as debuggers, symbolic execution engines, and emulators.

We first highlight important concepts adopted from avatar one, a hybrid rehosting tool. Then we provide an overview of `avatar`$^2$, discuss important details, and compare it to other frameworks which combine two or more tools for improving binary analysis.

### 4.3.1 A Bit of History: Avatar One

`Avatar`$^2$ is the successor of Avatar [ZBFB14], a system originally designed to rehost embedded devices for dynamic analysis, which we completely re-designed and extended to allow an easy orchestration of arbitrary components that can be combined to perform sophisticated binary analysis tasks. In essence, Avatar allowed partial emulation of firmware inside S2E, a symbolic execution engine based on QEMU. To achieve this goal, I/O requests which cannot be emulated are forwarded to the actual embedded device, either via dedicated debugging ports or by using a debugging stub manually injected into the device.

Although the purpose of the original tool was solely to enable dynamic binary analysis for embedded devices by connecting S2E [CKC11] to a physical device, it introduced a number of important concepts for building an orchestration framework suitable for coordinating multiple dynamic binary analysis tools. More precisely, it provided the following building blocks for a more general dynamic binary analysis orchestration framework:

- **Target Orchestration.** Avatar introduced the concept of orchestration, not simply as a way to control its two targets (S2E and the physical system), but also as mean to automatically transfer the execution from one tool to the other, based on certain events specified by the analyst.
- **Separation of Execution and Memory.** While the execution of a software and its memory space are tightly linked together in traditional analysis approaches, Avatar decouples them. This, among others, allows the framework to use a so called *remote memory*, whereby the execution proceeds on one

target, while memory reads and writes are forwarded to another target. This allowed Avatar to achieve partial emulation, whereas the main firmware is executed in an emulator, while accesses to memory-mapped peripherals are forwarded to the actual device.

- **State transfer and synchronization** Next to the orchestration of execution, Avatar provided the possibility to selectively transfer the state from one of its targets to another, where the state is defined by the combination of the content of the memory as well as the CPU registers. This allowed Avatar to execute initialization functions on the physical device under analysis, before transferring the state to S2E to perform symbolic execution.

### 4.3.2 General Overview & Terminology

Combining and connecting a variety of distinct tools requires a careful planned design to cope with the inherent challenges arising from the large diversity of tools. For example, such tools often use both asynchronous and synchronous communications.

On an abstract level, the `avatar`$^2$ framework consists of four distinct elements, as visualized in Figure 4.2. The `avatar`$^2$ *core*, *targets* and *protocols* are python libraries while *endpoints* are third-party software (such as other analysis frameworks, emulators, or solutions to talk to physical devices) controlled and interconnected by `avatar`$^2$.

The `avatar`$^2$ core has three purposes: i) to serve as the main interface for the analyst using the framework, ii) to carry out the actual orchestration and serve as interface to all the underlying elements, and iii) to catch, dispatch, and react to events generated by the various protocols while communicating with the respective endpoints.

Targets play the role of abstracting each endpoint and providing high-level interfaces to the `avatar`$^2$ core. However, these python abstractions do not directly communicate with their associated endpoints. In fact, since the actual communication often requires similar patterns that would otherwise be duplicated in multiple targets, it is mediated by a layer of specific *protocol* objects. This architecture makes individual protocols easy to reuse when prototyping new targets. This is for instance the case for a variety of debuggers and emulators that, while typically equipped with their own communication interface, often incorporate also a `gdbserver`, which can be controlled by gdb's *remote serial protocol*.

The protocols themselves are divided according to their purpose. In most of the cases, a target needs at least a memory protocol, an execution protocol, and a

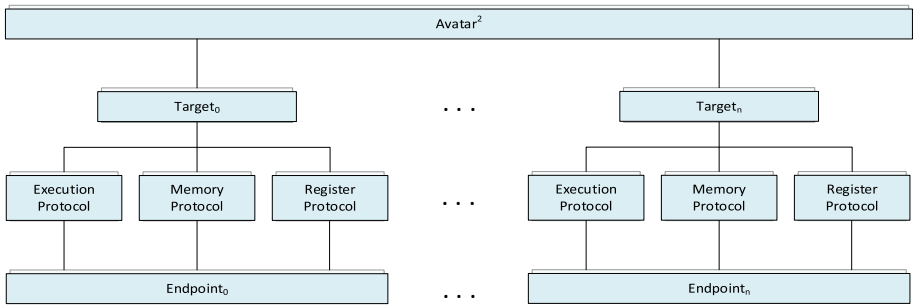


**Figure 4.2:** Overview of `avatar`$^2$.

register protocol. These protocols are responsible, respectively, for dispatching memory reads and writes, controlling the execution of the target, and accessing its CPU registers. Additionally, `avatar`$^2$ provides the possibility to define additional protocols, such as *monitor* protocols specifically dedicated to monitor the status of an endpoint or specialized *remote memory* protocols that can provide a custom high bandwidth channel for memory accesses from one endpoint to another.

Finally, *endpoints* can be anything worth orchestrating for an analysis, and the initial implementation of `avatar`$^2$ provides target abstractions for six different types of endpoints, which we will present in more details in Section 4.3.4.

The strict separation and abstraction of the different components allow a flexible configuration of a variety of different targets. Thus, in comparison to the first version of Avatar, the scope of the framework is extended far beyond the initial target of dynamically analyzing embedded devices firmware. This is due to the drastic shift of paradigm in `avatar`$^2$: instead of orchestrating specific tools with a specific goal, the core goal of `avatar`$^2$ is to enable a general interoperability among an arbitrary number of different tools frequently used for dynamic binary analysis.

### 4.3.3 Under the Hood

So far we introduced the general design of the framework. We now highlight and discuss in more details five specific features provided by `avatar`$^2$. These features are intended to provide additional flexibility in order to cope with different dynamic binary analysis tools.

### Architecture Independence

With the emerging interconnectivity of software not only on commodity computers, but also on embedded systems, the variety of architectures and instruction sets of interest for program analysis systems is broader than ever. Intuitively, as several dynamic binary analysis frameworks already come with support for multiple architectures, an orchestration framework should also be able to cope with those. `Avatar`$^2$ handles this problem by relying on a flexible description of the architectures in a modular manner, with the additional possibility to provide annotations for specific targets (i.e., special variables defined in the architecture that are fetched and consumed by targets). While the current implementation is shipped with descriptions for *x86*, *x86_64* and *ARM*, the modular approach allows to easily extend the framework to support additional architectures or even intermediate representations used by specific tools.

### Internal Memory Representation

`Avatar`$^2$ is designed to interconnect a variety of targets, which internally rarely use the same representation of memory. However, to synchronize the analysis across different frameworks and platforms, a consistent view of a program's memory is required. Hence, `avatar`$^2$ provides interfaces to the analyst for defining and updating the memory layout, which is then pushed to the targets. For instance, unlike other tools, `avatar`$^2$ does not represent memory on page granularity. This is because its goal to be able to cope with embedded devices which often consist of memory mapped peripherals and CPU registers that use only a fraction of a common page. Instead, `avatar`$^2$ works by combining *memory ranges* of arbitrary and non-uniform sizes.

### Legacy Python Support

The initial prototype of `avatar`$^2$'s core was written in Python 3.x. While we believe that a future migration to Python 3 is inevitable, several popular dynamic analysis frameworks, such as angr, manticore, and triton were either based completely on – or export bindings only to – python 2.7 at the time of `avatar`$^2$'s development. Therefore, we decided to work with Python 3 but still maintaining legacy python support to enable a flawless and performant integration to the aforementioned tools.

### Peripheral Modeling

Embedded devices often consist of custom peripherals which are not implemented inside other endpoints or – even worse – that cannot be represented equivalently in other endpoints at all. Like the first Avatar, `avatar`$^2$ is able to solve this issue

by memory forwarding. However, as memory forwarding can quickly become a performance bottleneck, avatar$^2$ provides an additional way to face this issue by adding prototypes of simple peripherals models. These models can be easily developed in python by the analyst and are, in essence, simple objects that respond to memory reads and writes at specified offsets. Facilitating these modeling mechanism, avatar$^2$ provides for instance an implementation for a universal serial port interface (USART) which models an interface present in a particular ARM based microcontroller by STMicroelectronics. Hereby, the model receives and transmits input and output over a Transmission Control Protocol (TCP) connection, instead of a physical peripheral.

**Plugin System**

Avatar$^2$ is designed to have a minimalistic and easy-to-maintain core, whereas the more complex logic is provided by the specific targets and protocols. However, a variety of tasks required by most dynamic analysis procedures are repetitive and therefore would be very inefficient if the analysts would need to re-implement them for each experiment. Therefore, to provide a common code base for these repetitive tasks and to execute them automatically, avatar$^2$ adopts a rich event-driven plugin-system. Within a plugin, various events can be hooked by custom callbacks, or completely new features can be added to the avatar$^2$ core. Examples for already existing plugins are an assembler and a disassembler, a forwarding plugin for single instructions, which is for instance useful to dispatch co-processor accesses, or a plugin for an *automated* orchestration of the analysis.

### 4.3.4 Supported Targets

Avatar$^2$ is designed to integrate new targets with low effort. Currently, it supports six targets, which already provide a large number of analyses combinations.

**The Gnu Debugger (GDB).**
The ability to communicate with GDB is probably one of the most essential features of avatar$^2$. The stand-alone target allows to debug GNU/Linux software. Moreover, a variety of endpoints are offering a gdbserver for debugging purposes. Due to the separation of targets and protocols, avatar$^2$ is able to communicate to all of these endpoints.

**OpenOCD.**
Modern CPUs and MCUs, and in particular those used in embedded devices, expose standardized debugging ports, such as Joint Action Test Group (JTAG) or Serial Wire Debug (SWD) ports. OpenOCD is an open source tool able to control debug dongles which can be attached to these ports.

Those dongles, together with OpenOCD, can be used for fine-grained debugging of the executed software. Naturally, `avatar`$^2$ supports OpenOCD to perform analysis of embedded devices.

**Quick Emulator (QEMU).**
As seen before, QEMU is a very popular emulator and forms the basis for many rehosting systems. At its core, the emulator uses dynamic binary translation to enable emulation of software written for different architectures. Although it allows to emulate single GNU/Linux programs in its user mode via dynamic system call translation, `avatar`$^2$ uses its full system emulation mode to handle hardware peripherals, too. We add two noteworthy components to QEMU which we maintain as part of the `avatar`$^2$ project. First, we introduced a new emulation machine, the *configurable machine*, which is able to configure the memory layout in a flexible way. This facilitates the integration with the memory representation of `avatar`$^2$ but also to support variety of embedded devices. Second, we added a set of dedicated avatar peripherals, which are responsible for the interaction with other targets during an analysis.

**PANDA.**
The Platform for Architecture-Neutral Dynamic Analysis (PANDA) [DGHH+15] is a dynamic analysis engine with focus on enabling repeatable reverse engineering. PANDA is based on QEMU, and hence, reuses the same components incorporated in the `avatar`$^2$'s QEMU target. The strength of PANDA lies on its additional capabilities to record, replay, and analyze a previously-recorded concrete execution.

**angr.**
The symbolic execution and program analysis framework angr [SWS+16] provides a number of powerful dynamic analysis capabilities. Several small additions to angr have been performed for a better integration with `avatar`$^2$. Those modifications are mainly on angr's state and memory management. More precisely, a special representation for memory pages to provide access to the memory of other targets has been added, together with several automatic procedures to set up an analysis state that can be used for `avatar`$^2$. Furthermore, as the angr target in `avatar`$^2$ inherits angr objects, no direct modifications of angr are required.

**Unicorn.**
Unicorn [QV15] is a lightweight, multi-architecture CPU emulator framework, which focuses solely on raw emulation of CPU code. The framework is based on QEMU, whereas all code which is not directly related to CPU

emulation is removed. As a result, Unicorn is distributed in the form of a single library, which exports bindings to various languages. `Avatar`$^2$'s Unicorn target does not require any modification of this library and uses its python bindings.

### 4.3.5 Comparison to Other Tools

Even though `avatar`$^2$ is - up to our knowledge - the first attempt to flexibly combine debuggers, emulators and dynamic binary analysis frameworks in a generic manner, a few tools have been directed to solve specific subproblems also tackled in `avatar`$^2$.

First and foremost, several existing tools embed or integrate other, third-party tools. Driller [SGS+16] for instance combines angr [SWS+16] with AFL [Zal14] to benefit from both the advantages of symbolic execution and fuzzing. Similarly, FrankenPSE [Tra16a], allows sharing of snapshots between PySymEmu [Man13], a symbolic execution tool and the GRR Fuzzer [Tra16b]. Unfortunately, as of time of writing, no open source version of FrankenPSE has been made available, which makes a direct comparison to other approaches difficult.

Independently, angr recently deployed a modular design-philosophy, allowing for exchanging different parts within the symbolic execution framework. As a result, different execution engines, IR or constraint solvers can be plugged into the framework.

Another example for a tool benefiting from external projects is *radare2* [Alv]. Although being a reverse engineering framework as its core it facilitates code emulation thanks to a custom IR. Nevertheless, it is designed to be enabled to control a variety of debuggers, and provide implementations for GDB and WinDBG. On top of this, community based plugins are integrating frameworks like Miasm [Des12] or emulators like Unicorn [QV15] into the radare2-ecosystem. Although this, just like angr, already enables powerful analysis, the specific tool is, together with its initial purpose, in the foreground.

Next to interconnecting independent targets, approaches for decoupling execution and memory have been frequently incorporated by partial emulation systems, for instance by Surrogates [KKM15] and Prospect [KPK14], which both use memory forwarding to enable the analysis of embedded devices (c.f. Chapter 4.1.2).

Additionally, a lot of modern dynamic binary analysis frameworks with different purposes are based on QEMU. DECAF [HPY+14] for instance focuses on just-in-time virtual machine introspection and tainting, while PANDA [DGHH+15] provides primitives for recording and replaying executions, whereas advanced analy-

sis plugins are used during the repeatable replay of an execution. Rev.ng [DFPA17], on the other hand, is most notably known for recovering control flow graphs and function boundaries as basic block for subsequent analyses and S$^2$E [CKC11] expands full-system emulation with the capability of symbolic and concolic execution. Furthermore, even tools for analyzing embedded devices firmware without having the actual device are based on QEMU, such as Firmadyne [CWBE16], which emulates a generic kernel for Linux-based firmware, and LuaQEMU [Ral17] which provides prototyping of hardware boards in Lua.

While all of those tools have their strength and are already quite powerful, they are rarely designed with interoperability in mind. As a result, the majority of those tools heavily modify QEMU for their purposes, effectively denying an easy integration with other tools. The patches for QEMU done by `avatar`$^2$, on the other hand, are minimalistic and centralized in the code base, which leads to an easy integrability of those tools as future targets for `avatar`$^2$.

### 4.3.6 Pitfalls & Gains of `Avatar`$^2$

In order to effectively combine and orchestrate different frameworks, `avatar`$^2$ needs to be generic enough to support a variety of frameworks with different design philosophies, execution primitives and scopes. Even though we think that it is infeasible to be generic enough to allow support for every dynamic binary analysis frameworks, we believe that our abstraction and distinction of targets, protocols and endpoints enable a variety of frameworks to be potential targets for `avatar`$^2$.

In fact, we designed `avatar`$^2$ to keep the implementation overhead for adding a new target as simple as possible. To add a new target, an analyst needs first to decide over which protocol instances it communicates to the associated endpoints. In case the endpoint cannot be controlled over already existing protocols, the implementation of the additional protocols will require the majority of the effort. However, we believe that by providing protocol implementations for both GDB and QEMU-based targets, already a decent amount of potential endpoints can be integrated into `avatar`$^2$ without the need to add new protocols. Once the right protocols are chosen, the actual target class can be written, which needs to provide interfaces for functionalities specific to the target, and an initialization function, which sets up the endpoint and connects the different protocols to it.

One of the main goals of the framework is to enable popular dynamic binary analysis frameworks to interoperate with embedded devices firmware. While two out of the three examples were targeting embedded devices, both were relying on the presence of a JTAG interface. Unfortunately, when analyzing real world hardware, such an interface is not always available. However, even in those cases `avatar`$^2$

can be used together with these hardware instances if, for example, a gdbserver can be launched directly on the device or a GDB stub can be injected at runtime, for instance using a bootloader. Unfortunately, those stubs are highly dependent on the architecture of the analysed target and are hard to abstract in a generic way. Although such stubs already exist for some targets[2] they are not integrated in `avatar`$^2$ yet.

An additional challenge for embedded devices is given when embedded devices do not communicate over MMIO, but trigger interrupts, e.g., upon arrival of new data. `Avatar`$^2$ provides an experimental support for forwarding interrupts on some hardware. However, the lack of genericity of interrupt handling is a limitation of the framework. Indeed, the way interrupts are triggered and served is tightly coupled to the hardware they are occurring on. As a result, interrupts need to be implemented in a per-target manner by using dedicated protocols for dispatching interrupts.

[2] https://github.com/avatarone/avatar-gdbstub or qcdebug [Del11]

# Chapter 5

# Enhancing Dynamic Analysis & Testing for Embedded Systems

In this chapter, we use the primitives provided by `avatar`$^2$'s multi-target orchestration to develop important additions to dynamic binary firmware analysis and testing. In more detail, we show how `avatar`$^2$ can be used to reproduce a previous study, we demonstrate the feasibility of record & replay for embedded devices, and adapt dynamic fault detection heuristics to embedded systems using the instrumentation capabilities provided by `avatar`$^2$'s PANDA target.

We then provide an experimental evaluation of fuzz testing embedded systems whereby we also demonstrate the effectiveness of our fault detection heuristics. Finally, we provide an outlook on concolic testing framework for firmware and discuss a prototype design and implementation.

## 5.1 Facilitating Replication and Reproduction

Recent initiatives are pushing more and more towards developing processes for facilitating reproduction[1] of scientific studies in the system security field [CP16]. However, when embedded devices are involved, reproduction of previous work is often complicated.

We choose HARVEY [GBC+17], a recently presented PLC rootkit as an example for reproduction. Using this example we show that `avatar`$^2$ can be used as a lightweight mechanism to prototype scripts for reproduction of previous studies

[1] Following the terminology from https://www.acm.org/publications/policies/artifact-review-badging

**Listing 5.1:** Re-implementation of HARVEY, using `avatar`[2].

```
from avatar2 import Avatar, ARMV7M, OpenOCDTarget

input_hook = '''mov r5,0xfffffffc
                b 0x20001E30'''

output_hook = '''mov r5,0xfffffffd
                 mov r4, r5
                 mov r5, 0
                 b 0x2000233E'''

avatar = Avatar(arch=ARMV7M)
avatar.load_plugin('assembler')

t = avatar.add_target(OpenOCDTarget,
        openocd_script='harvey.cfg',
        gdb_executable='arm-none-eabi-gdb')

t.init()

t.set_breakpoint(0xd270)
t.cont()
t.wait()

t.inject_asm('b 0x2000250E',addr=0x20001E2E)
t.inject_asm('b 0x20002514',addr=0x20002338)

t.inject_asm(input_hook, addr=0x2000250E)
t.inject_asm(output_hook,addr=0x20002514)

t.cont()
```

(or share scripts to facilitate reliable replication). This is a good example for a reproduction study, as this publication contains all the necessary details to reproduce it while no source code or scripts are publicly available.

The HARVEY rootkit is designed to be inserted in the firmware of an Allen Bradley PLC. It modifies the functions responsible for forwarding updates of physical inputs and outputs to other parts of the hardware, such as the LEDs and the Human Machine Interface (HMI). As a result, the rootkit is able to tamper with the PLC's I/O in a stealthy manner without reporting suspicious behavior on the LEDs or HMI. For deploying the required modifications, the authors of HARVEY describe two ways of deploying the compromised firmware on the PLC:

1. **Using JTAG debug access to patch the firmware in memory.** This is possible in this scenario, as parts of the firmware are loaded into and executed from the on-chip SRAM of the main MCU, which—by design—can be modified during run time.

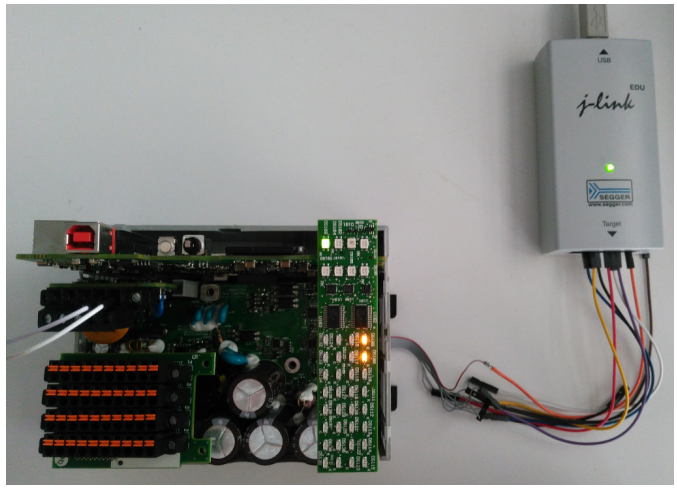


**Figure 5.1:** PLC Infected with HARVEY using `avatar`$^2$ as instrumentation framework. The communication with `avatar`$^2$ is performed with a JTAG debugger.

2. **Abusing the firmware update functionality to persistently upload the malicious firmware.** This requires to exploit the firmware update process. However, the authors did not follow this path as the firmware for this PLC is cryptographically signed and the installation of a modified firmware would either require the ability to create a new firmware with colliding SHA-1 hash, the knowledge of the PLC's manufacturers private key, or the presence of a flaw in this signature verification mechanism.

Although we use a slightly different version of the PLC than in the original[2], we were able to reproduce the base proof of concept implementation of HARVEY using `avatar`$^2$ and with about 30 lines of python (Listing 5.1). Figure 5.1 shows the PLC after infection by HARVEY: the two orange LEDs indicate the presence of input signals on Port 1 and 2, albeit no inputs are connected to the PLC.

This example provides several interesting insights into the `avatar`$^2$ framework. Line 12 loads the assembler plugin, which adds the capability to assemble and inject code into memory, as done in Line 24-28. Then a target is configured (Line 14-17), and it is initialized as a standalone target (Line 18). However, the hooks for HARVEY can not be inserted right away, as the secondary firmware code has first to be loaded into the SRAM. Hence, we insert a breakpoint after those initialization functions (Line 20), before starting the execution (Line21) and waiting for the

[2]We used an Allen Bradley 1769-L16ER-BB1B CompactLogix 5370 PLC, while HARVEY was initially implemented on an Allen Bradley 1769-L18ER-BB1B CompactLogix 5370

breakpoint to be hit (Line 22). The hooks are then inserted (Line 24-28) and the execution is resumed (Line 30).

While the full rootkit can be injected into the firmware without the presence of `avatar`$^2$, we want to stress that the usage of the framework provides a *centralized* and *unified* interface for dynamic instrumentation, which greatly eases replicability and, in turn, reproducibility.

## 5.2 Recording and Exchange of Firmware Execution

Traditionally, dynamic analysis of firmware for embedded devices requires either the presence of the physical device or the ability to emulate the firmware. In this section, we show that `avatar`$^2$ is capable of recording parts of the execution of a firmware by partially emulating it. This recording can then be replayed and analyzed without the presence of the physical device. To achieve this, we instruct `avatar`$^2$ to use the `PandaTarget` to emulate and record the core parts of the firmware, while memory accesses to hardware peripherals are forwarded to a physical device, controlled by `avatar`$^2$'s `OpenOCDTarget`.

In this example, we use a Nucleo STM32L152RE development board as physical target, which comes with an ARM Cortex-M3 MCU whose JTAG connection is available over the on-board USB interface. We use an example firmware form *ARM mbed*[3]. As often with embedded code, it performs many low-level memory accesses to the hardware's peripherals. PANDA alone would not be able to emulate the execution of this firmware, because the firmware would not execute properly without proper emulation of the device's specific peripherals.

The corresponding `avatar`$^2$ script is shown in Listing 5.2 and exhibits a couple of notable points. Line 3-12 are responsible for the usual general setup of `avatar`$^2$ and targets. Note that this time the orchestration plugin is loaded, which allows a different way to control the execution of targets as seen in the previous example. Line 14-18 define an *explicit* memory layout, including the ROM memory, backed by the firmware, and MMIO, which will be forwarded to the physical device, using the *remote memory* functionalities of `avatar`$^2$.

After the definition of memory, the targets are initialized (Line 20) and the actual orchestration is set up: Line 22 defines which target shall be used for execution first. Line 23 adds *transition*, in which `avatar`$^2$ will switch the execution from one target to another, while synchronizing the registers and memory ranges specified by the `synced_ranges` argument. In this case, only the RAM range needs to be synchronized, as this is the only dynamic memory local to more than one tar-

[3] https://www.mbed.com

**Listing 5.2:** Recording an embedded device's firmware execution.

```
1  from avatar2 import  ARMV7M, Avatar, OpenOCDTarget, PandaTarget
2
3  avatar = Avatar(arch=ARMV7M)
4  avatar.load_plugin('orchestrator')
5
6  nucleo = avatar.add_target(OpenOCDTarget,
7            openocd_script='nucleo-l152re.cfg',
8            gdb_executable='arm-none-eabi-gdb')
9
10 panda = avatar.add_target(PandaTarget,
11           executable='panda/qemu-system-arm',
12           gdb_executable='arm-none-eabi-gdb')
13
14 rom = avatar.add_memory_range(0x08000000, 0x1000000,
15         file=firmware)
16 ram = avatar.add_memory_range(0x20000000, 0x14000)
17 mmio= avatar.add_memory_range(0x40000000, 0x1000000,
18         forwarded=True, forwarded_to=nucleo)
19
20 avatar.init_targets()
21
22 avatar.start_target = nucleo
23 avatar.add_transition(0x8005104, nucleo, panda,
24          synced_ranges=[ram], stop=True)
25 avatar.start_orchestration()
26
27 panda.begin_record('panda_record')
28 avatar.resume_orchestration(blocking=False)
29
30 [...]
31
32 avatar.stop_orchestration()
33 panda.end_record()
```

get within this analysis. The last argument, `stop`, instructs `avatar`$^2$ to stop the orchestration after this transition. The reason for this state transfer lies in the desire to execute the initialization functionalities of the physical device on the device itself, as they are not of interest for the analysis of the main firmware.

The following line (Line 24) starts the automatic orchestration. `Avatar`$^2$ will run until a transition with the stop flag is hit, which is trivial in this case, as only one transition is defined. As a result, Line 27, which enables the recording of execution inside PANDA, is executed after the transition finished and the automatic orchestration has to be resumed (Line 28). Once the interesting parts of the firmware's execution have been recorded, both the orchestration and the recording can be stopped (Line 32-33). Afterwards, the execution recording is available and can be reused in PANDA (e.g., to perform further analysis) without using the embedded device.

This demonstrates the importance of separation between execution and memory for multi-target orchestration, as it allows the forwarding of MMIO in the first place. Without this, the initial recording, even with the presence of the physical device, would not be possible as long the underlying hardware platform cannot be fully emulated.

### 5.2.1 State Caching for Partial Emulation

Another interesting primitive for dynamic analysis is state caching for partial emulation setups. In essence, `avatar`$^2$ state transfer capabilities allow to save the device state after it is initialized, which then can be used to initialize an emulator.

In fact, since the initialization procedures of embedded systems usually set up all peripherals, this phase is rarely of interest for dynamic analysis, as user interaction is rather uncommon in this stage. Moreover, initialization procedures involve a large amount of I/O operations that have a negative impact on the performance of a partial emulator (that needs to forward every access to the physical device).

However, as long as later peripheral interaction only concerns *stateless* peripherals, this overhead can be removed by taking advantage of the ability of emulators to execute from an initial snapshot. Therefore, dynamic testing can benefit from the ability of `avatar`$^2$ to save a snapshot of the device state after its initialization and reuse it later to initialize the emulator.

## 5.3 Dynamic Binary Instrumentation for Fault Detection

Unfortunately, while common desktop systems have a variety of mechanisms to detect faulty states (e.g., segmentation faults, heap hardening and sanitizers) and to analyze them (e.g., command-line return values or core dumps), embedded devices often lack such mechanisms because of their limited I/O capabilities, constrained cost, and limited computing power. As a result, *silent memory corruptions* occur more frequently on embedded devices than on traditional computer systems, creating a significant challenge for conducting dynamic testing on embedded systems software, as demonstrated in Section 3.3.

Hence, we use the primitives provided by multi-target orchestration to dynamically instrument firmware and implemented six fault detection heuristics mimicking existing compile-time, and run-time techniques. While the underlying algorithms of the heuristics are based on known principles, we implemented them as external run-time monitors for embedded systems. We use PANDA to emulate the firmware and rely on its plugin system to obtain feedback over the execution of a partial or fully emulated firmware. All this while `avatar`$^2$ orchestrates the execution and selectively redirects execution and memory accesses to the physical device.

More specifically, we implemented the heuristics as PANDA analysis plugins, as the emulator allows to hook various events, such as physical accesses to memory, translation, and execution of translated blocks. During those hooks, the state of the emulator is available to the analysis plugin, allowing it to access concrete registers or memory access values. Additionally, the base implementation of PANDA provides already several analysis plugins[4], making the emulator a perfect building block for run-time instrumentation.

During run time, we then use `avatar`[2] to orchestrate the execution of the firmware on a target embedded device and on PANDA. This orchestration setup allows the possibility to automatically transfer state between device and emulator and forward I/O accesses while our instrumentation checks for the occurrences of memory corruptions. Additionally, the setup enables not only to perform analysis *during* the execution, but also to create lightweight records of the execution which can be used for later analysis with more heavy-weight instrumentation.

The heuristics themselves are inspired by techniques that have already been in use for detecting or mitigating memory corruptions in other settings, such as shadow stacks, compiler warnings for unsafe function calls, and run-time verification as implemented by instrumentation tools like Address Sanitizer [SBPV12] or Valgrind [NS07].

However, it is also important to mention that we selected heuristics to be implementation independent, in order to not only work in a live analysis setting (as it is the case if the firmware is run in an emulator) but also to be applicable "post-mortem" on previously recorded runs of the firmware, or even collected execution traces (in case that hardware-based tracing mechanism are available). As a result, they only rely on information extracted from the execution of the binary code and additional annotations provided by the analyst. In particular, our six heuristics are:

**Segment Tracking:**

Segment tracking is possibly the simplest technique that aims to detect illegitimate memory accesses. The core idea is to observe all memory reads and writes and verify if they occur in valid locations, thus somehow imitating an MMU at detecting *segmentation faults*. This technique only requires knowledge about the memory accesses and the memory mappings of the target,

[4] In fact, we build our instrumentation heavily on the `callstack_instr` plugin. This plugin allows registering further callbacks on function calls and returns, and provides information about the current call stack, which simplifies the implementation of several of our heuristics. However, as this plugin would return wrong information when the state of the application is corrupted, we only use its `on_call` event to detect the beginning of a function, while information of function returns are retrieved by analyzing the executed blocks.

which is easily accessible in an emulator. Additionally, the memory map can be obtained by reverse engineering when only traces from the execution are available.

**Format Specifier Tracking:**
Tracking format string specifiers is a naïve technique to discover insecure calls to `printf()`-style functions and is inspired by the printf protection outlined in [She17]. In essence, this protection validates that the format string specifier points to a valid location upon entry in a function of the `printf()` family. In the simplest case, without presence of dynamic generated format string specifier, those valid locations would have to lie within read-only segments. All in all, this technique requires not only knowledge about the location of format handling functions, but also the register state while entering one of those functions and the argument order. Both the location of the according function and their argument order can be obtained by reverse engineering or automated static analysis of the firmware.

**Heap Object Tracking:**
This technique is designed to detect both temporal and spatial heap related bugs and is influenced by the instrumentation and run-time verification approaches presented in [SBPV12]. It achieves its goal by evaluating the arguments and return values of *allocation* and *deallocation* functions and bookkeeping the location and sizes of heap objects. This allows to easily detect out-of-bounds memory accesses or access to a freed object. However, this heuristic depends on a variety of information: executed instructions, the state of the registers, memory accesses, and knowledge about allocation and deallocation functions. The latter could be retrieved by reverse engineering, or by using advanced methods for discovering custom allocators as demonstrated by MemBrush [CSB13].

**Call Stack Tracking:**
This heuristic is replicating traditional shadow stack protections [BZP19], therefore aiming at detecting functions that do not return to the callee. This can help to identify stack-based memory corruptions that overwrite the return address of a function. It does so by monitoring all direct and indirect function calls and return instructions. However, as embedded devices are often interrupt-driven, this heuristic can lead to false positives.[5] Nevertheless, it is especially appealing as it requires the least amount of information: only the knowledge of the executed instructions.

[5]We want to note that corresponding false positives did not occur in our later experiments, as we tested interrupt-free parts of the firmware. In any case, these false positives can be ruled out by refining the heuristic to detect the presence of interrupt contexts.

**Call Frame Tracking:**

Call frame tracking is a more advanced version of the call stack tracking technique which detects coarse grained stack-based buffer overflows, without false positives, right when they occur. In essence, stack frames are located by tracking function calls, then contiguous memory accesses are checked not to cross stack frames. Hereby, this requires to identify the executed instructions as well as register values to extract the stack pointer values upon function entries. Then, memory accesses have to be observed to detect the actual corruptions.

**Stack Object Tracking:**

Stack objects tracking consists in a fine-grained detection of out-of-bound accesses to *stack variables*, which is inspired by the heap object tracking approach proposed by Serebryany et al. [SBPV12]. Hereby, memory reads and writes observed during execution are checked against the individual variable size and position in the stack. Obviously, this requires to track executed instructions and memory accesses, as well as elaborate information about the stack variables. For the sake of simplicity, we use variables information which is present in debug symbols. However, in the general case, it is possible to retrieve such information in an automated manner from binary code, as proposed by several previous studies [SSB11, JCG+14].

Table 5.1 lists the six presented heuristics and details what type information is required for the analysis. We want to stress that those heuristics are just one example of dynamic binary firmware instrumentation enabled by multi-target orchestration. We choose to implement them as an example to tackle the Challenge-3, fault detection, by turning silent memory corruptions into observable ones.

| | Execution | Register state | Memory Accesses | Memory Map | Annotated Program |
|---|---|---|---|---|---|
| Segment Tracking | ✗ | ✗ | ✓ | ✓ | ✗ |
| Format Specifier Tracking | ✓ | ✓ | ✗ | ✓ | ✓ |
| Heap Object Tracking | ✓ | ✓ | ✓ | ✗ | ✓ |
| Call Stack Tracking | ✓ | ✗ | ✗ | ✗ | ✗ |
| Call Frame Tracking | ✓ | ✓ | ✓ | ✗ | ✗ |
| Stack Object Tracking | ✓ | ✓ | ✓ | ✗ | ✓ |

**Table 5.1:** Implemented fault detection heuristics and their requirements.

## 5.4 Fuzzing Embedded Systems: An Experimental Evaluation

In recent years, fuzz-testing has become more and more popular as a way to test the security of embedded systems, but is still not as widely adopted as for software targeting desktop computers.

In this section, we aim to understand the difficulties for fuzz testing embedded systems and perform an experimental study to analyze how fuzz testing firmware can benefit from the runtime analysis techniques integrated in our multi-target orchestration framework.

### 5.4.1 Past Experiments

For understanding the challenges of fuzz testing embedded devices, we first analyse recent literature. Table 5.2 provides a short overview of recent efforts in the area of fuzz testing embedded systems' firmware. These works covered different sectors, different input generation strategies, and different classes of embedded devices.

For instance, Alimi et al. [AVR14] fuzzed parts of the MasterCard M/Chip specifications by generating test-inputs using a genetic algorithm. The authors observed that real cards would become unusable during comprehensive fuzz-testing, and therefore moved the parts under test into a simulator where they could trigger the approval of illegitimate transactions.

Some modern smart cards also contain a web server implementation. Kamel et Lanet [KL13] designed a generation-based fuzzer for the HTTP implementations of those web servers and were able to trigger several flaws, including errors in the smart cards. Koscher et al. [KCR+10] conducted a security test of automotive systems using fuzzing in addition to reverse engineering and packet sniffing. More

| Study | Sector | Fuzzing Approach | Type of Device | | |
|---|---|---|---|---|---|
| | | | Type-I | Type-II | Type-III |
| [KCR+10] | Automotive | Mutation-based, Blackbox | - | - | ✓ |
| [MGS11] | GSM feature phones | Generation-based, Blackbox | - | ✓ | - |
| [KL13] | SmartCards | Generation-based, Blackbox | - | - | ✓ |
| [ABSZ14] | PLCs & SmartMeters | Mut.- & gen.-based, Blackbox | - | ✓ | - |
| [AVR14] | SmartCards | Generation-based, Blackbox | - | - | ✓ |
| [VDBHT14] | Smartphones | Generation-based, Blackbox | ✓ | ✓ | - |
| [KPK14] | – | Mutation-based, Dynamic Inst. | ✓ | - | - |
| [LCC+15] | Automotive | Random-based, Blackbox | - | - | ✓ |
| [TTZ+18] | Smartphones | Mutation-based, Greybox | ✓ | - | - |
| [ZDY+19] | Routers | Mutation-based, Greybox | ✓ | - | - |

**Table 5.2:** Fuzzing experiments of embedded systems in the literature.

specifically, the authors fuzzed packets for the CAN bus, discovering packets to unlock all doors, disable the car's light, and permanently enable the horn. Similar results were reported in a later study by Lee et al. [LCC+15]. Hereby, random fuzzing of the data field in CAN packets led at least to observable changes in the car's instrumentation panel.

In [ABSZ14], Almgren et al. developed several mutation-based and generational-based fuzzers which were used to test various PLCs and smart meters. The fuzzing experiments lead to the discovery of several known and unknown denial of service vulnerabilities, some leading to a completely unresponsive PLC which could only be recovered after a power cycle and a cold restart.

Mulliner et al. [MGS11] and Van den Broek et al. [VDBHT14] developed a generation-based fuzzer in accordance to the GSM-specification to test GSM implementations in both feature- and smart-phones. While both studies were able to trigger a large number of errors – including memory exhaustions, reboots, and denial-of-service conditions – the authors concluded that correctly monitoring the devices under test in an automated manner is still a very challenging task.

To avoid this problem, systems are sometimes fuzzed under partial emulation. The authors of PROSPECT [KPK14], for instance, could discover a previous unknown 0-day vulnerability in a fire alarm system by fuzz testing different network protocol implementations and monitoring the state of the partially emulated system.

Similarly, Charm [TTZ+18] fuzzes partially emulated android device drivers using Syzkaller [Vyu15], a coverage-guided, greybox fuzzer. The authors ported the source code for android device drives to x86 and run the fuzzing campaign against a virtualized environment, whereas I/O operations are forwarded to a physical phone. While the idea of cross-architecture fuzzing is interesting, this approach relies on the presence of source code for the fuzzed components.

Yet another approach utilizing coverage-guided greybox fuzzing for binary firmware is presented by FirmAFL, which adapts the popular fuzzer American Fuzzing Lop (AFL) to fuzz applications on Type-III devices by smartly switching between user-mode and full-system emulation. Unfortunately, the presented approach is deeply tied to the abstraction provided by the operating system and can not easily be applied to Type-II or Type-I systems.

### 5.4.2 Core Challenges for Fuzzing Embedded Devices

Fuzzing can be performed with or without the availability of the source code. Source code availability makes testing more efficient as memory semantics can be used to detect anomalies, for example, by using compile time corruption de-

tection techniques. However, fuzzing embedded devices—which lack memory protections, exploit countermeasures, and for which source code is very rarely available—becomes quite difficult.

Based on our experience and on problems reported by other authors while conducting fuzzing experiments in previous works, we believe that three specific challenges, out of the ones we identified in Chapter 3, are the limiting factors for fuzzing embedded systems:

1. **Fault Detection.** Most fuzzing techniques are relying on *observable crashes*, and while desktop systems offer protection measurements which are triggering a crash upon a fault, embedded devices are often lacking according mechanisms.

2. **Scalability.** Fuzzing greatly benefits from multiple instances of the software under test. While this is easy achievable for desktop systems, it would require the availability of multiple devices for embedded systems.

3. **Instrumentation.** In recent years, a variety of instrumentation techniques for aiding fuzzing have been developed. Unfortunately, they often rely on primitives not available when fuzzing embedded systems, such as advanced operating system features or recompilation of source code.

On top of these specific problems, which make fuzzing for embedded systems much more complex and less efficient compared to desktop systems, fuzz testing firmware also inherits other challenges which are common to fuzzing in general (e.g., test case generation) but we consider them out of the scope of this work.

Furthermore, the combination of lacking fault detection mechanisms and missing instrumentation capabilities complicates the detection of memory corruptions triggered by the fuzzer. Indeed, testing parties often aim to identify a successful memory corruption by monitoring the device to detect signs of an incorrect behavior. As a result, sophisticated *liveness* checks (or *probing*) are commonly deployed to check the effects of single test cases on the device. In general, it is possible to adopt two types of probing. *Active probing* inserts special requests into the communication to the device or to its environment. This influences the state of the software under test – as the state of the software is periodically checked by providing *valid input* and evaluating the corresponding response. In contrary, *passive probing* aims at retrieving information about the device's state without altering it. This could be achieved by looking at the answers provided by the device to the test inputs or by observing *visible crashes*.

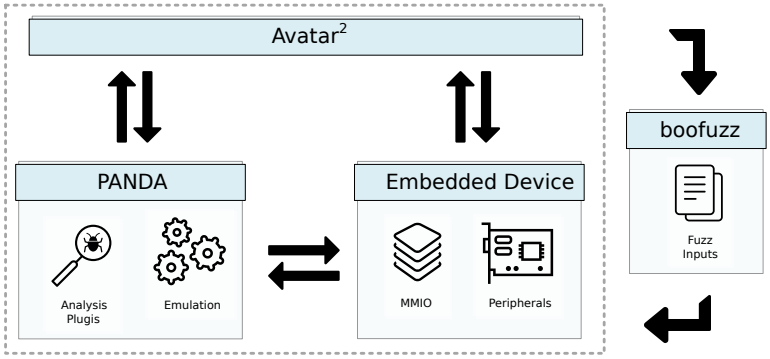


**Figure 5.2:** Setup of our fuzzing experiments.

However, as we showed in Section 3.3 relying on visible crashes can be misleading as many corruptions will remain *silent* in the context of embedded systems. Hence, "liveness" checks are insufficient to detect many classes of vulnerabilities as we will show in the remainder of this section.

### 5.4.3 Experiment Setup

We conducted a number of experiments to test the benefits of multi-target orchestration for fuzz testing. More specifically, we want to assess how the fault detection heuristics presented in Section 5.3 perform when fuzz testing a partially or a fully emulated device while comparing it to a traditional device-only fuzz test. The basic setup of our experiments is visualized in Figure 5.2 and will be described more in the following.

Our goal is to show that it is possible to integrate our heuristics in a live fuzzing experiment, thus providing fault detection to mitigate the lack of equivalent mechanisms in the embedded system's platform and operating system. However, our approach may introduce a non negligible overhead on the performance of the system, effectively increasing the time required to perform the testing session. Therefore, we also decided to measure the effects of our solution on the fuzzing throughput under different setup configurations.

#### Target Setup

For our tests, we compiled the very same vulnerable *expat* application used in Section 3.3 to demonstrate the severity of lacking fault detection mechanisms for a Type-III device[6].

[6]Note that we chose a Type-III device because this is the most challenging case. The intuition is that if we can detect the silent memory corruptions on a Type-III device, the heuristics are likewise suitable for Type-II and Type-I devices.

We then conducted a number of fuzzing sessions against the target using four different configurations, covering both optimal and worst case scenarios:

- **NAT: Native.** Fuzzing is performed directly against the actual device – therefore without using any fault detection heuristic. We use this case as the baseline to compare the results of other experiments.
- **PE/MF: Partial Emulation with Memory Forwarding.** The firmware is emulated and access to the peripherals is implemented by forwarding I/O memory accesses to the actual device.
- **PE/PM: Partial Emulation with Peripheral Modeling.** The firmware is emulated and peripheral interaction is handled by leveraging the peripheral modeling capabilities of `avatar`$^2$, which allows to conduct experiments without having a physical device present.
- **FE: Full Emulation.** Both the firmware and its peripherals are fully emulated inside PANDA.

For configurations *PE/MF*, *PE/PM*, and *FE* we use a snapshot of the device's state taken *after* initialization as starting point for the emulation, as described in Section 5.2.1. The execution then continues inside the emulator where we implemented the different heuristics presented in Section 5.3. To estimate the performance impact imposed by each scenario, we conducted experiments in which we selectively enable one heuristic at a time.

In all our tests, the inputs to the vulnerable software are provided on a simple textual protocol which is communicated over a serial connection. On configurations *NAT* and *PE/MF* we used the real device serial port, while in *PE/PM* and *FE* the serial port of the device is either modeled or emulated, and the input is provided to the emulator with a TCP connection and written directly in the corresponding (emulated hardware) buffer.

**Fuzzer Setup**

We built our experiments on top of `boofuzz` [Per16], an open source fuzzer and successor to Sulley [AP07], which is a popular Python-based fuzzing framework. Boofuzz does not only generate and send malformed inputs, but it also allows target monitoring and defining reset hooks. In comparison to its predecessor, it allows to fuzz over user-defined communication channels and provides readily available implementations for both serial and TCP-connections, making it an ideal match for our evaluation purposes.

To make sure that the results of different experiments are comparable, we instrumented the fuzzer to forcefully generate inputs which would trigger one of the inserted vulnerabilities with a given probability. We denote this probability as $P_c$ and conducted experiments with $P_c = 0$, $P_c = 0.01$, $P_c = 0.05$ and $P_c = 0.1$.

Furthermore, to better simulate a realistic fuzzing session, we added a simple liveness check for monitoring purposes: after every fuzz input, the fuzzer receives the response of the device and evaluates whether it matches the expected behavior. When the received response differs from the expected one, or when the connection times out, the fuzzer reports a crash and restarts the target. The fuzzer uses `boofuzz` to power cycle the physical device (*NAT*) or instructs the emulator to restart from the snapshot (*PE/MF*, *PE/PM*, and *FE*).

Note that we use the liveness checks during all experiments, even when all heuristics are enabled as there might be crashes not detected by our heuristics. However, in our experiments, with all heuristics enabled, the liveness check never detected any corruption because the heuristics of our PANDA plugins were able to detect faulty states at a earlier stage, which in turn triggered an immediate restart of the target.

### 5.4.4 Results

In total, we conducted 100 fuzzing sessions lasting one hour each. We monitored the number of inputs that were processed by the target ($I_{tot}$), the number of times a corrupting input was sent to a target ($I_C$), the amount of faults detected by the liveness check ($D_L$), and the number of faults detected by the heuristics ($D_H$). Additionally, we denote the number of undetected faults as ($D_U$). As a result:

$$I_C = D_L + D_H + D_U \approx I_{tot} * P_C \tag{5.1}$$

#### False Positives and False Negatives

Intuitively, the presented heuristics are not perfect and are likely to yield false positives or negatives. Interestingly, we observed only one case of false positives in the stack object tracking when, due to over-approximation, two consecutive memory writes to set two distinct but adjacent stack variables were falsely tagged as a memory corruption.

In general, we want to stress that false positives and negative rates are highly target- and implementation-dependent and a comprehensive analysis of those are out of scope of this work. Our goal is to show the limitations of fault-detection on embedded devices and the feasibility of using heuristics to overcome this problem, rather than evaluating the effectiveness of a specific implementation.

### Fault Detection

Table 5.3 shows which type of corruptions could be detected by the liveness check or by the individual heuristics. As we expected, fuzzing without any fault detection mechanism is largely ineffective. The liveness check alone was only able to detect the stack-based buffer overflow and format string vulnerability because, as we already described in Section 3.3, these bugs result in the device hanging. All the other vulnerabilities, although they were triggered correctly by the fuzzer and they resulted in a successful memory corruption, were not detected by the fuzzer.

The impact of this is shown in Figure 5.3, which visualizes the amount of corrupting inputs detected by the liveness check, by the heuristics (all[7] or in isolation), or that remained undetected. A closer look at the graphs shows that the combined heuristics (rightmost bar in each group) always successfully detected all corruptions, while relying on liveness checks (leftmost bar) always left a large fraction of faults undetected. Furthermore, segment tracking, as it can both detect format string and stack based buffer overflow vulnerabilities, is superseding all detections formerly done by the liveness check. This makes sense: when the device is in a strongly corrupted state, even detectable by the liveness check, it is likely that memory accesses to unusual memory locations occurred.

| | Format String | Stack-based buffer overflow | Heap-based buffer overflow | Double Free | Null Pointer Dereference |
|---|---|---|---|---|---|
| Liveness Check | ✓ | ✓ | ✗ | ✗ | ✗ |
| **Individual Heuristics:** | | | | | |
| a) Call Stack Tracking | ✗ | ✓ | ✗ | ✗ | ✗ |
| b) Call Frame Tracking | ✗ | ✓ | ✗ | ✗ | ✗ |
| c) Stack Object Tracking | ✗ | ✓ | ✗ | ✗ | ✗ |
| d) Segment Tracking | ✓ | ✓ | ✗ | ✓ | ✓ |
| e) Format Specifier Tracking | ✓ | ✗ | ✗ | ✗ | ✗ |
| f) Heap Object Tracking | ✗ | ✗ | ✓ | ✓ | ✓ |
| **All** | ✓ | ✓ | ✓ | ✓ | ✓ |

**Table 5.3:** Artificial vulnerabilities discovered by the different heuristics.

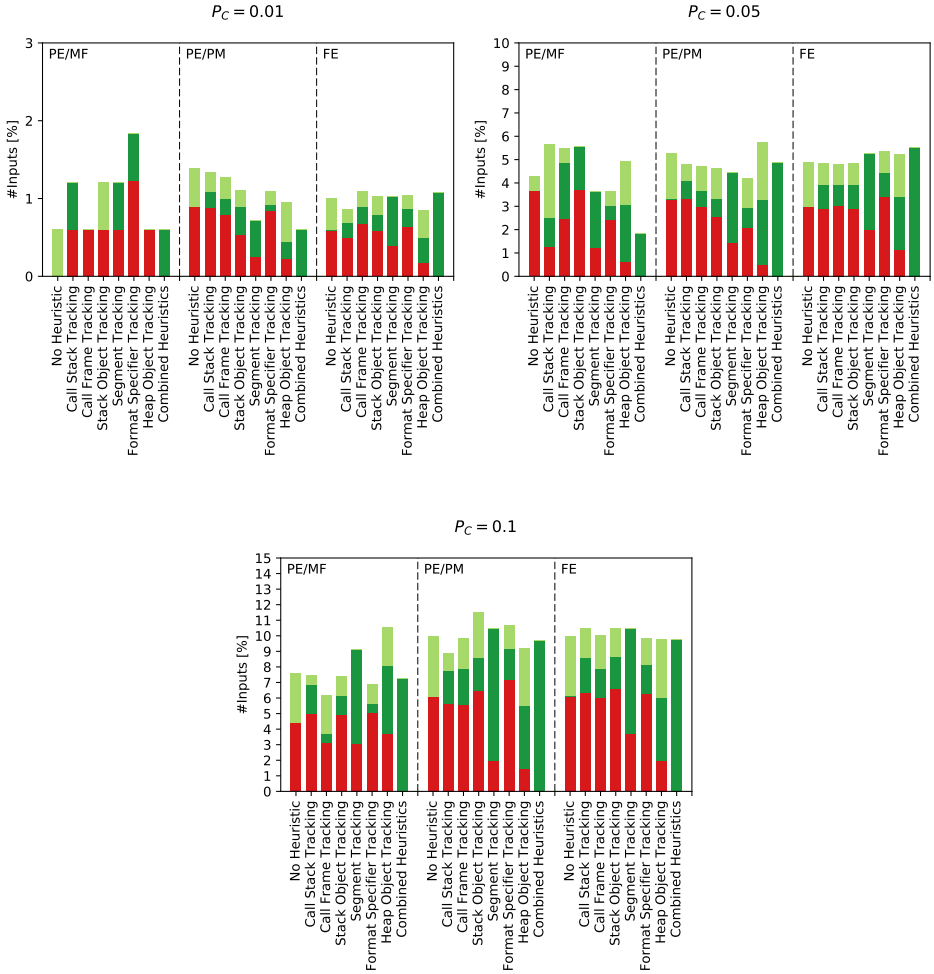


**Figure 5.3:** Corruption detection in emulation based scenarios with distinct probabilities for the occurence of corrupting inputs $P_C$.
Corruptions detected by liveness checks
Corruptions detected by heuristics
Undetected corruptions

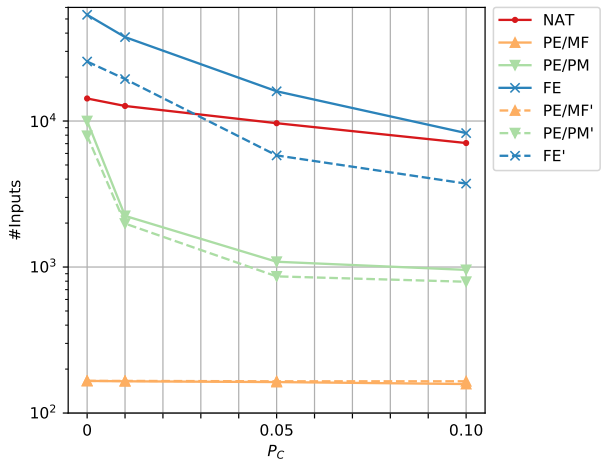


**Figure 5.4:** Processed inputs during one hour long fuzzing sessions with no heuristics (*NAT*, *PE/MF*, *PE/PM*, *FE*) and combined heuristics (*PE/MF'*, *PE/PM'*, *FE'*) enabled.

### Performance

Figure 5.4 shows the number of input values the fuzzed target was able to process during one hour fuzz sessions with different values for $P_C$. As expected, partial emulation with memory forwarding (*PE/MF*) is slowing down fuzz testing by more than one order of magnitude. This overhead is introduced by the communication between the firmware and the device peripherals, which results into frequent invocations of the orchestration features of Avatar. The major part of this overhead is due to the low bandwidth connection between Avatar and the physical device, which relies on a standard JTAG debugger connected via USB. Surrogates [KKM15] has shown that this issue can be solved by using dedicated hardware, which would enable partial emulation at near-realtime speed.

Looking at the individual heuristics, we can observe that their overhead is negligible in the *PE/MF* scenario, where the bottleneck of forwarding of MMIO requests fully determines the speed of the fuzzing experiment. However, in scenario *PE/PM* and *FE*, we can can observe a considerable slowdown (between x1.5 and x6) introduced by the heuristic analysis code for $P_C = 0$.

[7]Note that the "Combined Heuristics" consist of heuristic *c-f*. Heuristic *a* and *b* have been disabled as they are redundant with heuristic *c*.

Another important observation is that fuzzing against a fully emulated target is significantly faster than against the physical device, as long the amount of detected corruptions is low. This is due to three main factors. First, the fact that the communication over TCP allows a higher throughput than the one over a serial port. Second, by the fact that even if the firmware is emulated, the emulator often has a much higher clock speed than (low resource) embedded devices. Third, a detected corruption is tied to a forceful reboot of the target, which means that high $P_C$s are resulting into significant time spent rebooting, rather than sending new inputs to the target.

However, the most important result of our experiments is the fact that a firmware that is executed in PANDA (full emulation) with combined heuristics enabled can be fuzzed faster than the original embedded device under realistic values for $P_C$. While the first can detect all classes of vulnerabilities we inserted in its code, the second needs to rely on a liveness check that can only identify two of them.

### 5.4.5 Outcome & Interpretation

The results of our experiments show that silent memory corruptions pose a predominant challenge for fuzzing embedded systems, as the majority of fuzzing solutions are relying on observable crashes. In particular, our tests emphasize three different aspects:

1. **Relying only on liveness tests is a poor strategy.** Fuzzing embedded systems by relying solely on liveness tests for fault detection is a poor strategy that is very likely to miss many vulnerabilities. Likewise, using only a single heuristic at a time does not guarantee the detection of more vulnerabilities. Intuitively, the highest potential for corruption detection is reached by combining several heuristics.

2. **While full emulation is the best strategy, emulators are rarely available.** Our experiments shows that, if it is possible to fully emulate the firmware of the device under test, then few selected heuristics can mitigate the lack of fault detection mechanisms. This increases the accuracy of vulnerability discovery to what we now expect when fuzzing desktop applications. While this may seem to solve the problem, full emulation is still very difficult to achieve. In particular, third party testers often lack sufficiently detailed knowledge of the hardware to implement a good emulator. Moreover, even with sufficient documentation, implementing a full emulator requires a considerable amount of manual effort.

3. **Partial emulation can lead to accurate vulnerability detection, with a significant performance impact.**
   When full emulation is not possible, partial emulation can lead to the same benefits in term of accuracy, at the cost of a significant slowdown of roughly one order of magnitude. In particular, partial emulation with peripheral modeling provides an interesting trade-off between vulnerability detection and fuzz speed throughput and does neither require a sound emulator nor a physical device to be present. Moreover, this setup allows parallelizing fuzzing sessions, thus making fuzz-testing embedded devices more scalable.

A further advantage of our emulation-based approach is that PANDA could also be used to record and replay the execution, which largely simplifies the followup analysis to identify the root cause and possible impact when a vulnerability is detected.

Another interesting observation is that liveness checks often detect crashes due to a timeout, which significantly slows down the fuzzing experiment. In an optimal setup, where live heuristics are able to detect the majority of corruptions, the liveness check could be omitted, which could result in a significant performance improvement. This is conceptually demonstrated in Figure 5.5 for a simplified scenario where processing the liveness check and processing the fuzz-input is taking the same computation time.

Finally, while our results directly impact the performance of fuzzing embedded systems, this work also applies to binary symbolic execution on embedded devices firmware (e.g., as described in [ZBFB14]). An important problem of symbolic execution is the state explosion problem: with a sufficiently complex program and symbolic input the symbolic execution can rapidly reach a very large number of states which exhaust resources or takes indefinitely long time to complete. Typically, state exploration will continue until a terminating condition is found. Therefore, if the corruptions are not promptly detected, the symbolic execution could spend a significant amount of time computing useless states.

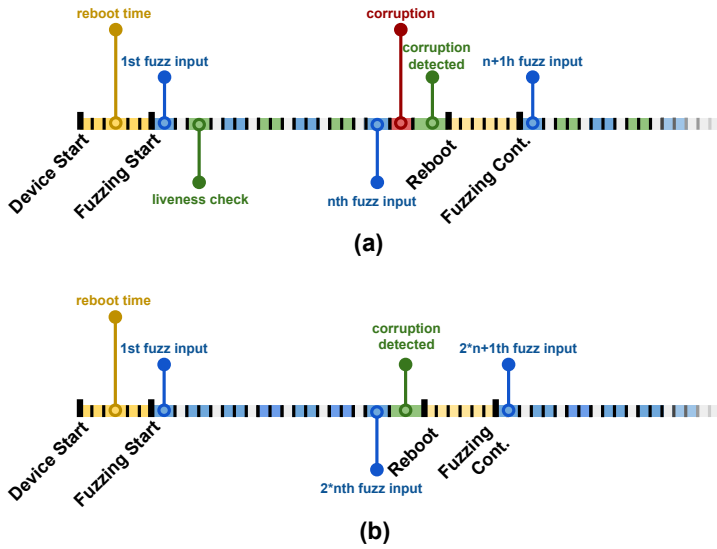


**Figure 5.5:** Example timelines of a fuzzing session with (a) liveness checks, and (b) live detection without liveness checks.

## 5.5 Outlook: Concolic Testing on Firmware

Offline symbolic execution confines common problems of symbolic execution by only analysing one execution path at a time, trading efficiency with operationality [BCD+]. One particular interesting kind of offline symbolic execution is *concolic execution*, a mixture of *concrete* and *symbolic* execution, in which a symbolic execution engine follows traces from concrete runs of the program under test. Although concolic execution have been shown to be valuable for testing desktop software [BGM13], it has not been widely adopted as testing method for binary firmware.

In this section, we will analyse the state of the art of symbolic and concolic execution for firmware and propose a novel, concolic execution based approach.

### 5.5.1 Prior Art

Similar to fuzz testing, both symbolic and concolic execution gained traction as testing technique for firmware recently. Besides the targeted firmware type and the used execution engine, previous studies mainly differ in two key aspects: whether they require the presence of source code, and whether they embed the actual device in the analysis. Table 5.4 shows a selection of previous studies and the corresponding key aspects.

While approaches requiring source code are not applicable for binary firmware analysis, they help to understand the intrinsic challenges for symbolic execution on firmware. For instance, Kim et al. [KKJ12] used CREST [BS08], an open source concolic test generation tool for C code, to concolically execute various applications for Type-III embedded systems. They point out that it is easier to use source code in this context, as a binary-based approach would require symbolic execution engines to implement the specific hardware platform for the software under test, which requires a considerable amount of effort. This is in line with the challenge of platform variety, as pointed out in Section 3.1.2.

| Study | Engine | Binary Based | Uses Device? | Type of Firmware | | |
|---|---|---|---|---|---|---|
| | | | | Type-I | Type-II | Type-III |
| [KKJ12] | CREST | - | - | ✓ | - | - |
| [DMRJ13] | KLEE | - | - | - | ✓ | ✓ |
| [ZBFB14] | KLEE (S2E) | ✓ | ✓ | - | ✓ | ✓ |
| [SWH+15] | angr | ✓ | - | ✓ | ✓ | - |
| [CZJ+15] | SMAFE | ✓ | ✓ | - | ✓ | - |
| [HFT+17] | KLEE, angr | ✓ | - | - | ✓ | ✓ |
| [CCF18] | KLEE | - | ✓ | - | ✓ | ✓ |

**Table 5.4:** Symbolic and concolic execution on firmware: exemplary studies.

FIE [DMRJ13] demonstrates that this challenge is even exacerbated when symbolically analysing Type-II and Type-III systems, where interaction with the hardware is frequently directly integrated in the program under analysis, as chips and peripherals now need to be modeled. Additionally, for those systems, interrupts can potentially occur at any time, which requires symbolic execution engines for source and binary based analysis alike to implement special means for interrupts to avoid state explosions. FIE for instance requires the analyst to provide an InterruptSpec, while FirmUSB [HFT+17] uses a dedicated interrupt scheduling mechanism.

Another way to deal with this problem is presented by Inception [CCF18], where a physical device is used and occurring interrupts are forwarded to the symbolic execution engine in a tightly synchronized manner. Besides overcoming the issue of interrupts, invoking the device has another additional advantage for symbolic and concolic analysis on binary firmware: concrete state is present and can be used to populate the symbolic execution engine, as done in avatar one [ZBFB14]. This is directly opposed to binary-based, no-device approaches as Firmalice [SWH+15], in which the state is over approximated and requires additional optimizations such as *lazy initialization*.

Summing up this analysis, we believe that a flexible concolic execution engine for binary firmware should use the physical device because it (a) prevents the need of implementing new emulators for every tested device, and (b) offers concrete state for bounding symbolic exploration to a single path. (c) limits the problem of state explosion through hardware interactions.

However, up to our knowledge, only one study performs concolic execution on embedded systems following this approach. Chen et al. [CZJ+15] use run time information collected by dedicated remote debugging features provided by VxWorks to guide SMAFE, a symbolic execution engine originally developed for windows executables [CZZ+13]. Unfortunately, besides being coupled to VxWorks, the tool makes a 1-to-1 relationship between symbolic execution engine and analyzed device, which is severely limiting the scalability of this approach.

### 5.5.2 The Terrace Testing Platform

In this chapter, we will present a novel approach for Testing Embedded systems using Record & Replay and Concolic Execution (Terrace). The core focus for this approach is to enable *scalable*, binary-only testing analysis which is achieved by a carefully crafted architecture.

### Overview

The basic idea of Terrace is seemingly simple: Decoupling the symbolic analysis and concrete execution on the device allows for sequentially running test cases on the device while parallelizing the slow, symbolic execution. Additionally, the record and replay functionality provided by `avatar`$^2$ further enables to generate traces for a symbolic execution engine during replay. As pointed out in Section 2.3.3, this is better than live trace generation, as a smaller run-time performance overhead is imposed on the embedded system.

Figure 5.6 illustrates the architecture of Terrace. As its core, Terrace consists of three different types of virtualized containers, which communicate with each other and use a Data Share holding important information for all three of them, such as configuration scripts or the binary firmware. The three container types are:

1. **Record Container.** Every container of this type is directly associated with a physical device and creates recordings of the execution of the partially emulated firmware under test.

2. **Analysis Container.** Containers of this type are replaying the records while generating traces compatible with the used symbolic execution engine.

3. **Backend Container.** Containers of this type perform the actual symbolic analysis and generate new test cases.

Concolic testing with Terrace is designed to be automated. As a first step, an initial test case is provided to a record container. Once the record is finished, it is added to the record queue. From there, it is passed on to an analysis container, which generates a trace, which in turn is passed to a backend container via another queue. This backend container then follows the trace symbolically while solving constraints and generating new test cases.

From here, the loop is repeated. However, as symbolic execution is slower than concrete execution, and as multiple test cases are generated at once, testing can now be parallelized: The record container sequentially executes all new test cases, while analysis and backend container asynchronously process generated records and traces. Additionally, thanks to the containerization, parallelized processing of records and traces is facilitated, as additional containers can be spawned on demand.

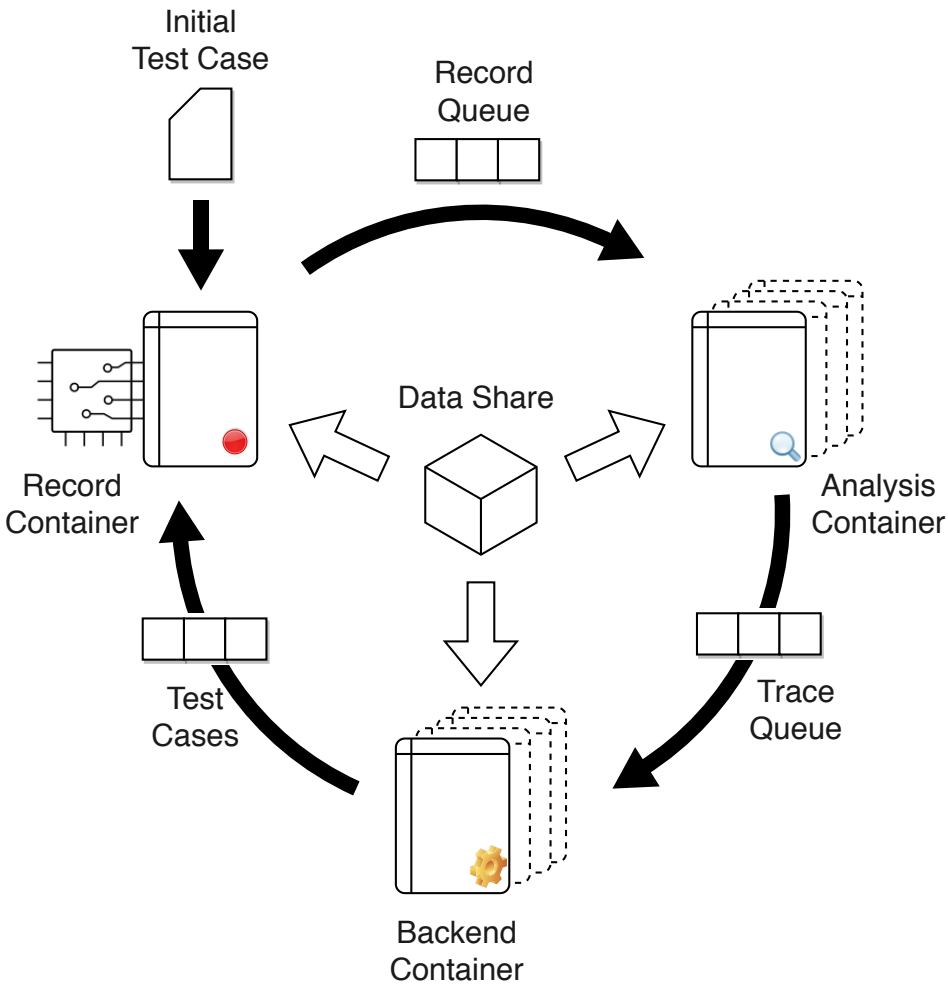


**Figure 5.6:** Overview of Terrace.

### Implementation

We built Terrace on top of Docker for providing containers and provide a single python script as interface for the analyst. The Data Share is realized via a shared volume exposed to all containers, whereas records and traces are exchanged via volumes only exposed to the corresponding container types.

The generation of records is performed using `avatar`$^2$, and an analyst has to provide an `avatar`$^2$-script capable of generating a record. It is noteworthy, that we implemented Terrace in a way that it can handle *multiple* record containers at the same time, effectively allowing concolic execution with a *device pool*. Obviously, this requires multiple physical devices, and each record container is associated with one of them.

The trace generation is performed with the help of PANDA plugins which log executed basic blocks and accessed memory. However, memory accesses can fall into different categories and are treated differently:

1. **Normal Memory.** As the name suggests, these accesses are plain, normal accesses. Whatever is written into a normal memory location will not change, and reads will always return the last-written value. Hence, Terrace only needs to log the values on those locations when they are first read.

2. **Special Memory.** Following the terminology of FIE [DMRJ13], MMIO has to be considered as special memory, as the content of this memory is driven by the peripheral, rather than the CPU. Hence, reads can return *different* values every time and are not necessarily returning data of preceding writes. Terrace logs every read to special memory in the trace so that the values can be replayed to the symbolic execution engine.

3. **Symbolic Memory.** This memory is simply the source for the symbolic data used later on in the symbolic execution engine. However, in comparison to traditional symbolic execution, two reads from the same location may return different symbolic values in the case of special memory. As a result, Terrace logs every byte read from symbolic memory, but flags it as symbolic. It additionally stores the concrete value, which allows for later concretization in case the symbolic execution needs to be bounded later on.

The information about which memory locations fall into which category is provided by the analyst in the global configuration. We designed this configuration to be compatible with `avatar`$^2$, so that the recording scripts can use this information for automatically setting up a partial emulation environment.

The last part of the implementation consists of the backend container. We designed Terrace to be flexible and, hence, allow for different symbolic execution engines as backend. At the time of writing, we implemented angr and KLEE as backends, because those engines are the most popular choices for symbolic analysis on firmware, as shown in Table 5.4. We adjusted both engines to operate on traces generated by Terrace, and in case of the KLEE backend, we additionally use PANDA during the analysis phase to generate the LLVM bitcode for execution inside KLEE. Additionally, before registering a new test case to the test case queue, Terrace ensures that it has not been enqueued, or tested, in a previous run of the loop.

**Discussion**

So far, our implementation for Terrace is solely experimental and a thorough evaluation for performance and efficacy is remaining for future work. Nevertheless, we believe that the approach presented by Terrace is promising, especially as it overcomes a set of common problems experienced for symbolic execution on firmware.

First of all, following concrete traces greatly simplifies symbolic exploration on embedded systems. Common problems like interrupt handling, or circumventing the creation of an non-manageable amount of states due to other hardware interactions are not necessary thanks to the benefits of offline symbolic execution.

Additionally, we allow for a huge amount of parallelization and even provide the possibility to distribute test cases among multiple embedded systems. Up to our knowledge, we are the first to propose the usage of a device pool combined with decoupled concolic execution, leading to a scalable testing method only bounded by the number of available devices.

Furthermore, by decoupling symbolic execution and trace generation, we allow for running additional analysis passes on the traces before passing them to the symbolic execution engine. For instance, embedded devices often spent considerable time in loops waiting for the arrival of new data or interrupts. Those loops can be easily detected, either manually by the analyst or by static analysis, and removed from the trace, which allows speed-ups for the symbolic execution.

Another benefit resulting from the modular design of Terrace is the possibility to easily exchange components. Instead of generating records, for instance, live traces could be collected using hardware-debug features, if present. Subsequently, the analysis container would not replay a record for trace generation, but format the collected traces to a Terrace compatible format.

# Chapter 6

# Firmware Unpacking Revised

For dynamically analyzing and testing firmware, one must first obtain a copy of the devices' firmware. However, as pointed out with Challenge-1, firmware retrieval, extracting firmware directly from a device is often a challenge on its own and may require sophisticated methods and dedicated equipment.

Another way for firmware retrieval are firmware update packages either directly obtained from the vendor, or intercepted during the update process [VOC18]. However, even when an update package is found, understanding its structure is often complicated, as data, binary code, and even full file systems are interleaved, frequently only organized by a vendor specific format. The process of understanding those formats and splitting the firmware image into its constituent parts is known as *firmware unpacking* and is a widely acknowledged challenge [CZFB14, SWH+15, LZL+16, SCA+18, SMB+18].

This chapter aims to provide more background on the topic of firmware unpacking and proposes Groundhogger, a novel system for firmware unpacking based on dynamic analysis. Furthermore, we present three case studies to exemplify the typical applications and use cases of Groundhogger.

## 6.1 A typical firmware update

As introductory example, we provide pseudo-code for a typical firmware updater in Listing 6.1. In most scenarios, the main firmware update routine combines the tasks of retrieving the update file, checking its validity, unpacking it, and transferring it to non-volatile storage. Errors could occur at any stage and the update routine normally returns early to avoid deploying a corrupted firmware update.

**Listing 6.1:** Typical Firmware Update Routine.

```
int update_firmware {
    void *update_file, *update_unpacked;

    update_file = get_firmware_update();
    if (update_file == -1)
        return -1;

    if (!verify_metadata(update_file))
        return -2;

    if (!verify_integrity(update_file))
        return -3;

    update_unpacked = unpack_firmware(update_file);
    if (update_unpacked == -1)
        return -4;

    if (!flash_firmware(update_unpacked))
        return -5;

    reboot();
}
```

The way the update file is retrieved depends heavily on the type of device. For instance, it could be delivered as a file on an external storage media, directly downloaded from the vendor's website, or simply manually transferred by the user over the network.

After the update is loaded into RAM, the update routine first ensures that it is a valid firmware update file by verifying that the file itself is in the right format and checking fields in the header, such as target device and version number. Should all those fast checks succeed, the integrity of the update file's content is validated. Most of the times, this is simply done by computing a checksum over the content and comparing it against a value given in the update file header. The checksum algorithms used vary and are often custom-tailored by the vendor. Recently, many vendors have begun using standardized signature schemes which additionally prove the authenticity of the update file. This also requires the public key of the vendor to be deployed on the device.

Next, the firmware update is unpacked. In simple cases (e.g., when the firmware is neither encrypted nor compressed), unpacking can be as simple as splitting the data into the different segments. However, many vendors rely on compression mechanisms to reduce the size of the update file and some choose to encrypt their firmware updates. Quite often, the used compression and encryption schemes are vendor-specific.

Once the firmware file is unpacked, the update routine flashes the new version to the non-volatile storage of the device, for instance NVRAM or flash memory. In most cases, the device is rebooted afterwards to finalize the deployment of the firmware update.

## 6.2 Unpacking in Prior Research

There is a vast amount of motivations for unpacking firmware next to dynamic testing, such as deploying patches to end-of-life systems, understanding the inner workings of the hardware platform, finding or inserting backdoors, or statically analyzing the firmware for vulnerabilities. To better understand how unpacking has been addressed in prior work, we reviewed recent work on firmware and embedded device analysis. Table 6.1 summarizes our review and provides several interesting insights.

In the academic literature, we find that no existing work has focused solely on the problem of unpacking; in fact, most of the time unpacking just forms the initial step for vulnerability discovery on firmware images. On the other hand, in cases where the goal was modification of the firmware, understanding of the update file format was obtained by manual reverse engineering. Additionally, the main corpus of work for large scale analysis is conducted on Linux-based firmware, most often running on routers or IP cameras. These kinds of devices appear to be "low hanging fruit," as firmware images can be easily unpacked with existing tools, in particular because they are often simple archives or standard file systems [LZL+16].

However, the scope of embedded devices goes far beyond Linux-based firmware. For example, Costin et al. [CZFB14] reported that 86% of their *unpacked* firmware is Linux-based. Yet only around 20% of their collection of firmware images were unpacked: Linux-based firmware is easier to unpack. The vast majority of large-scale unpacking efforts fails to unpack more than half of the firmware updates in the data set and focuses the subsequent analysis on successfully unpacked Linux-based firmware. It is possible that a large part of not unpacked firmware images are composed of monolithic firmware, encrypted or obfuscated images, which cannot be generically unpacked without significant reverse engineering effort, as for instance shown in [Cap16].

We conclude that the state of the art of automated unpacking focuses mostly on Linux-based firmware, leading to an over-representation of devices with easily-unpacked firmware. Unpacking for other types of firmware as common on Type-II and Type-III devices, however, has not been automated and is hence based on manual reverse engineering.

| Study | Motivation | Method | Unpacked (success/attempts) | Targeted devices |
|---|---|---|---|---|
| [ZKB+13] | Firmware Modification | Reversing | 1 / 1 | Harddrive |
| [CCS13] | Firmware Modification | Reversing | 373 / ? | Printers |
| [MEMS14] | Firmware Modification | Reversing | 1 / 2 | Mouse |
| [CZFB14] | Vulnerability Discovery | BAT (mod.) | 33,356 / 172,751 | Various |
| [CWBE16] | Vulnerability Discovery | Binwalk (mod.) | 9,486 / 23,035 | Various/Linux |
| [FZX+16] | Vulnerability Discovery | Unspecified | 8,126 / 33,045 | Routers, IP Cams, APs |
| [Cap16] | Vulnerability Discovery | Reversing | 1 / 1 | Undisclosed |
| [TGC17] | Backdoor Detection | Binwalk, FMK, BAT | 7,590 / 15,438 | Various/Linux |
| [LFW+18] | Device Fingerprinting | Binwalk | 5,296 / 9,716 | Routers, Gateways |
| [DPY18] | Vulnerability Discovery | Binwalk | ~2,000 / ~5,000 | Routers, IP Cams |
| [CLW+18] | Vulnerability Discovery | Binwalk | 6 / ? | Routers, IP Cams |

**Table 6.1:** Previous studies which use firmware unpacking.

### 6.2.1 Unpacking Tools

In terms of software for firmware unpacking, we found that most previous work relies either on the Binary Analysis Tool (BAT) [HKVD11], or Binwalk [Hef13].

BAT was originally designed to detect GPL violation via string matching and computing similarities between binaries and compressed data. As intermediate step, the tool recursively unpacks and decompresses data in common archive and compression formats identified by their magic numbers. Recently, the successor of the tool, Binary Analysis Next Generation (BANG) [Hem18], added support for a wider variety of unpackable file formats and included new features to add contextual information to unpacked files [Hem19].

Binwalk's main feature is its ability to perform signature scanning inside firmware images. Additionally, recent versions also support recursive unpacking and entropy analysis. The tool also exposes a Python API for scripting and can be enhanced with custom plugins, making it a common choice for firmware unpacking.

Finally, a third tool, FRAK [Cui12], is sometimes mentioned in the literature. However, neither the source nor the binary code of the tool has been publicly released, and therefore its adoption and use is very limited.

Besides these generic frameworks, custom unpackers and repackers for firmware of a specific device are quite common and are often developed after successful reversing of a specific firmware update format, such as in Cui et al. [CCS13] and Zaddach et al. [ZKB+13]. The Firmware Mod Kit (FMK) [HC13] is a collection of a variety of such tools combined with Binwalk in order to allow convenient unpacking and repacking of firmware in common image formats.

### 6.2.2 Challenges in Firmware Unpacking

Generic firmware unpacking is an open and widely acknowledged problem, and the difficulties are attributed to a variety of different reasons [CZFB14, SWH+15, CDZ+18, SCA+18, CLW+18]. In general, we identify the following four core challenges in unpacking:

1. **Obtaining Firmware Updates.**

   A logical requirement for unpacking firmware is acquiring the firmware itself. Although some vendors release update files on their website, this is by far not the norm, and even in those cases, the updates may only be accessed by authorized users. Additionally, some vendors only ship generic update software, which downloads the specific firmware update for the target device on demand—in a process that is completely opaque to the user.

   Another recent trend, especially for embedded devices serving in the so called "Internet of Things (IoT)" is Firmware over-the-air (FOTA). Here, the user does not initiate a firmware update at all—instead, a central server, most of the time vendor controlled, pushes the firmware update to the device automatically.

2. **Custom File Formats.**

   Firmware updates are normally shipped as a single file to enable easy deployment. Unfortunately there is no standard format for a firmware update. Instead, each vendor typically creates their own format, which may include custom headers and data sections, manifests, checksums, and embedded digital signatures. For a given vendor, different products can have different file formats, but in many cases a vendor will use the same file format across multiple products. Groundhogger can take advantage of this and use a single extracted unpacker to extract multiple firmware versions across multiple products.

3. **Compression & Encryption.**

   To save disk space and bandwidth, firmware updates are usually compressed using standard compression algorithms such as gzip and lzma. However, in many cases these algorithms have been modified slightly in ways that render them incompatible with standard tools.

   Encryption and digital signatures are also widely used by vendors in an effort to keep the code and data of the firmware hidden and prevent unauthorized firmware modifications. Here, again, there is a great diversity in the algorithms used: for instance, a standard algorithm such as AES may be

used with a fixed key that is embedded elsewhere on the device [Sch16], while other vendors may opt to implement a custom, usually weak encryption scheme of their own devising [Cap16].

In some cases, these modified algorithms have been reverse engineered and incorporated into open-source tools such as Firmware Mod Kit [HC13], but the large number of variants that exist means that there are many firmware images that have no easy means of decompression and decryption.

4. **Diversity of target platforms.**

   Challenge-2 for firmware analysis in general, the diversity of hardware platforms, also applies to unpacking. Naturally, embedded devices give few to no interfaces for introspection, and the variety of different instruction set architectures, peripherals, and operating systems makes it infeasible to emulate firmware code without further preparation. As a result, all kinds of dynamic analysis approaches are more difficult to exercise in comparison to traditional desktop systems.

## 6.3 Groundhogger: A Framework for Semi-Automated Unpacking

As seen in the last section, the majority of previous work attempts unpacking on a large scale. To cope with a huge variety of firmware, tools rarely attempt to parse the packaging format but rather rely on static analysis techniques, such as file carving, entropy analysis and magic number matching. As a result, parts of the content in a firmware update file can easily be missed. This imprecise unpacking does not suffice for all kinds of firmware updates. In particular, targeted analyses for specific devices often need to fall back to manual reverse engineering for creating *precise* unpackers.

We believe that the tedious manual effort for creating precise unpackers is an overlooked problem and could benefit from an automated or semi-automated solution, which we propose in this section. Our work aims to bridge the gap between large-scale firmware unpacking with static tools and case studies with single firmware images. More specifically, we exploit the partial emulation and instrumentation capabilities provided by `avatar`$^2$ to improve the state of firmware unpacking through *dynamic analysis*. The key insight is that a device's firmware must know how to unpack and process its own updates. By recording the firmware update procedure and extracting the relevant unpacking code, the unpacking capabilities can be re-hosted on an analysis system. Because vendors reuse their firmware image format across different products, this re-hosted unpacker can then be used to

unpack firmware for other devices. Additionally, we will show that the extracted unpacker can be automatically analyzed for checksum or signature verification functions, allowing for *firmware modification*.

### 6.3.1 Overview

In the following, we will present *Groundhogger*, a novel system for identifying and re-hosting firmware unpacking and validation methods.

The core idea of Groundhogger is based on two independent observations: (1) the firmware deployed on an embedded device must know how to process its own update files, and (2) multi-target orchestration allow for sophisticated dynamic analysis during run time of the unpacking routines.

In particular, Groundhogger takes advantage of the partial emulation and analysis capabilities of `avatar`$^2$ to automatically identify and extract the crucial parts of firmware update mechanisms, i.e., the functions responsible for unpacking and validating a new firmware image. Once extracted and re-hosted, these functions can be executed as standalone tools without the physical device. For instance, this allows an analyst to easily create unpackers for custom firmware formats. As we will show, this approach is especially valuable because vendors tend to re-use the same firmware file format and packing mechanisms across different products.

Groundhogger also eases the development of firmware modification attacks, by drastically minimizing the amount of manual reverse engineering involved in preparing a well-formed modified firmware.

All this is possible because in our work we only need to emulate a small snippet of code which typically has no or limited hardware interactions. In other words, the amount of I/O requests to the actual hardware is small and typically, neither unpacking nor validation routines are interrupt-driven, thus preventing bandwidth or latency issues, which are some of the biggest challenges that affect partial emulation systems.

### 6.3.2 Approach

While the core idea is simple, the actual challenge lies in its automation. In fact, to improve the state of the art, we want our system to be able to go from a device that can be partially emulated to re-hosted routines with a minimal amount of manual interaction. In the following, we describe the main steps of Groundhogger's analysis, which are also visualized in Figure 6.1:

1. **Device Preparation.** The first step, which needs to be performed manually, consists of setting up the target device for partial emulation. A common so-

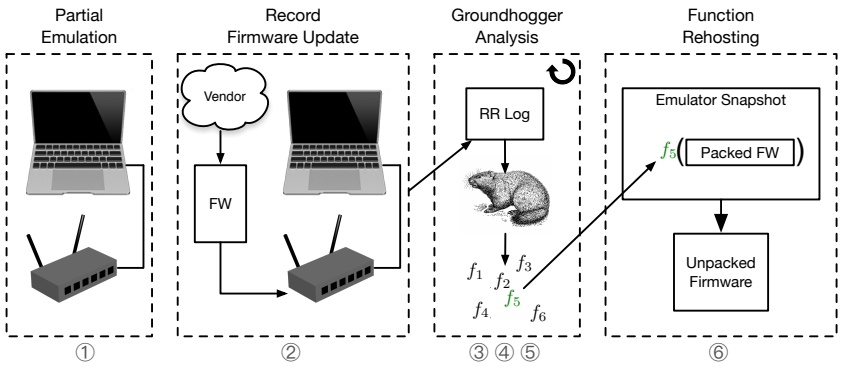


**Figure 6.1:** Overview of Groundhogger. The numbers below each stage refer to the steps in Section 6.3.2.

lution is to use debug interfaces such as JTAG [KKM15, CCF18, MNFB18], but other approaches exist that rely on injected stubs accessible via network [KPK14], USB [TTZ+18], or UART [ZBFB14].

2. **Trace Recording.** Once the device is ready, the firmware update has to be initiated. During the firmware update, the host emulator records the execution. Note that only the firmware update routine has to be executed inside the emulator, which largely simplifies the partial emulation process. This is the only phase in which partial emulation is required, as all following steps can be performed on the recorded information.

3. **Input Buffer Isolation.** This is the first step of Groundhogger's automated analysis. Its task is to locate the firmware update file in memory, denoted as `update_file` in Listing 6.1. Given that the firmware needs not be in memory at the beginning—or end—of the update routine, a simple linear search over the memory is often insufficient. Instead, Groundhogger dynamically locates the firmware update file by observing memory accesses during the replay of the recorded update. The location of the update file will serve as *input buffer* for the majority of functions during the update.

4. **Identification of Functions of Interest.** Groundhogger continues its analysis by identifying those functions that access the input buffer. In the example given in Listing 6.1, this would include the functions `verify_metadata`, `verify_integrity`, and `unpack_firmware`.

   The general intuition is that any function that processes the firmware update file can be of interest to the analyst. For instance, in most cases tagging functions which take a pointer to the input buffer as argument is sufficient

to find the unpacking routine. However, when this default behavior fails to locate the actual unpack routine, Groundhogger can also be instructed to tag all functions which perform at least one memory read access on the input buffer.

5. **Routine Selection.** Given the set of tagged functions, the analyst needs to select which ones she wants to rehost. In order to provide additional information to guide this selection, Groundhogger provides a flexible plugin system. For instance, at the time of writing we provide a plugin for automatically identifying decompression and decryption routines. It does so by first locating the output buffer for the unpacked firmware (`update_unpacked` in Listing 6.1) by tracking memory write accesses performed by the tagged functions. Locations which receive a large amount of continuous writes are very likely candidates for the output buffer. In a subsequent run of the replay, functions which modify the output buffer are identified and reported as targets for re-hosting, as they are very likely to perform the decompression or decryption routines.

6. **Function Rehosting.** The last step consists in re-hosting the desired routine together with the associated input, and, if applicable, output buffer. For this, Groundhogger creates first a snapshot of the emulator upon entry of the target function. Afterwards, it starts execution from there, while exchanging the input buffer with a firmware update file specified by the user.

   Once the function finished executing, Groundhogger can automatically extract and store contents of registers or memory to disk, which allows for instance retrieval of the unpacked firmware in case unpacking routines are rehosted.

While this methodology may also extract routines which are not easily rehostable due to hardware interaction (false positives), the user can reiterate over steps 4-6 until the correct functions are identified. Moreover, this iterative process can be easily automated by telling Groundhogger to simply attempt to rehost all the identified functions.

Additionally, Groundhogger provides valuable information for further reverse engineering, if required. In fact, the location of input and output buffers, as well as the set of functions of interest identified in step 4 are useful assets for understanding the details of a firmware update routine.

### Entry Point for Recording

A major challenge in our approach is how to determine where, in the firmware execution, the recording should start. This is important to identify which part of the code needs to be executed in the emulator, rather than on the physical device. In an ideal setting, the record should *only* contain the target routine, but if we knew it beforehand we would not need Groundhogger in the first place. Hence, as a good approximation, we try to identify the firmware update routine or one of its callees, and select it as starting point. This has two benefits: On the one hand, the update routine must necessarily unpack and validate the firmware update file, and on the other hand, the hardware interactions during the update routine are likely to be less frequent than during normal execution.

For identifying the update routine, one can always fall back to manual reverse engineering. Most of the times, the update routine is easy to identify by analyzing string references, as most firmwares print or log at least some kind of status information about the update. At the other end of the spectrum, if one has access to a perfect partial emulation system or a complete emulator for the hardware of the target device, one could simply record the complete execution starting from boot to the end of the firmware update, without the need to first locate a suitable entry point.

One heuristic to semi-automatically locate the entry point is to manually start the firmware update on the device and interrupt the execution shortly after. As unpacking, check-summing, and writing the updated firmware from volatile to non-volatile memory takes a considerable amount of time, it is likely that the execution would stop during the update process. From here, a full snapshot of memory and register state can be performed, and the firmware update routine can be found by analyzing the stack frames, which can be performed in an automated manner.

### Implementation Details

We implemented a proof-of-concept version of Groundhogger using `avatar`$^2$ for orchestrating embedded devices and PANDA [DGHH+15]. We use the emulator for recording and analyzing the firmware update procedures. As pointed out in Section 2.3.3, the advantage of relying on recordings is that they only store non-deterministic inputs and a snapshot of the initial state, which leads to a smaller memory footprint than conventional execution traces.

We developed a PANDA plugin that can automatically perform steps 3 to 5 in our approach, and use `avatar`$^2$ not only for partial emulation in the first step, but also to orchestrate the full process. Note that, even though our tool is based on PANDA, identification of the routine to rehost can also be performed on traditional

execution and memory traces gathered from the actual hardware, in case those features are available for the device under analysis.

### 6.3.3 Limitations

At a first glance, the solution presented by Groundhogger may appear unnecessarily complicated: why should we bother to setup a partial emulation system to unpack and repack firmware for a device we already have access to? The answer lies in the fact that Groundhogger aims to facilitate *rehosting* of functions of interest. This allows, among others, for the creation of re-usable and flexible unpackers: On the one hand, those unpackers can now be run without the physical device present which results into additional scalability. On the other hand, as we will show in the next chapter, unpacking routines are often re-used across different products of the same vendor, broadening the applicability and reusability of unpackers generated by Groundhogger.

Because Groundhogger is based on partial emulation [ZBFB14, MNFB18], it inherits the limitations of that technology. Specifically, partial emulation cannot currently handle devices that make use of Direct Memory Access (DMA), or devices that have no debug access. As partial emulation technology improves and starts supporting a wider range of devices, Groundhogger will also increase its ability to extract a wider range of update routines.

Groundhogger is also currently still a prototype, and requires manual intervention at some stages. Specifically, finding the entry point for the recording and starting the individual steps described in the last section has to be carried out manually. However, we believe that many of these steps can be automated in future work. We also want to stress that once a firmware unpacking routine has been successfully extracted by Groundhogger, it can usually be applied to other firmware versions for the same target device (or even to other devices from the same manufacturer), *without* manual intervention and without the need of acquiring any new hardware, as long the unpacking code does not rely on dedicated hardware features.

Another limitation for Groundhogger is firmware using *strong* cryptographic primitives in their update routine. In particular, devices using per-device keys and, subsequently, per-device updates are protected against the re-hosting approach presented in this work. In fact, while Groundhogger can still analyze and rehost the firmware unpacking and validation routines, these are not helpful for unpacking or modifying the firmware of *other* devices of the same type. However, Costin et al. [CZFB14] showed that reuse of cryptographic keys in embedded devices is–unfortunately–a widespread problem and our intuition is that firmware update keys are no exception to this rule.

For some targets it may be feasible to obtain an unpacked firmware image, but difficult to actually run the device under partial emulation (for example, one may not actually have the physical hardware at all). In this case it may be desirable to develop static equivalents of the heuristics used by Groundhogger to locate and extract firmware update functions. We leave this to future work.

An important advantage of Groundhogger is that the tool is, conceptually, not limited to firmware unpacking and validation routines. In fact, every function with limited hardware interaction could be identified and re-hosted when heuristics for tagging the function can be established. One promising example for this are, for instance, cryptographic primitives. Gröbert et al. have shown that they can be identified using dynamic analysis [GWH11] and Groundhogger can be easily extended accordingly. Another additional use-case could be the identification and re-hosting of parsing routines similar to what discussed in PIE [CZV+15], which in turn would allow for additional dynamic analysis techniques.

Overall, we see great potential for the approach presented by Groundhogger not only for firmware unpacking, but also for enabling different aspects of dynamic firmware analysis in the future.

## 6.4 Groundhogger: Case Studies

Now, we present three case studies carefully chosen to emphasize both the advantages offered by Groundhogger, and the challenges encountered in real world scenarios. These three case studies reflect different goals, as well as three very different challenges as summarized in Table 6.2.

The first example shows how to use Groundhogger to unpack a firmware packed with a very complex routine. The second shows how to use Groundhogger to prepare a modified version of a firmware, and how to cope with an embedded device running a Linux operating system. Finally, the third example shows how to unpack a firmware of a solid state disk drive, and how to carry out a modification attack when the code is cryptographically signed.

| Case | Device Type | Vendor | Model(s) | Motivation | Core Challenge |
|---|---|---|---|---|---|
| (I) | Digital Camera | Canon | EOS 60D/450D | Unpacking | Complex Update Procedure |
| (II) | IP Camera | Foscam | FI8918W | Modification | Complex Operating System |
| (III) | Solide State Disk | Micron | Crucial MX100 | Unpacking<br>Modification | Complex Hardware Platform<br>Signed Firmware Update |

**Table 6.2:** Summary of Groundhogger's case studies and their challenges.

### 6.4.1 Case I: Firmware Unpacking

#### Hardware Platform and Device Information

The first case study is purposefully selected to demonstrate the capabilities of Groundhogger under perfect conditions, i.e., when a device's firmware can be entirely rehosted in an emulator. We selected firmware updates for Canon EOS Digital Single-Lens Reflex (DSLR) cameras as our target, as an independent QEMU fork for emulating the firmware exists as part of the *Magic Lantern* project. To perform the recording on the firmware from a real device, we acquired a Canon EOS 60D camera.

#### Challenges

The main challenge of this first scenario is the fact that the update mechanism is unusually complex and involves multiple stages, which are summarized in Figure 6.2. We now provide the low-level description to showcase the intricacies of a firmware update routine and to establish a basis for understanding the results of Groundhogger. However, we want to stress that this information is not necessary for setting up and running Groundhogger.

In the first step, the firmware update file needs to be copied to an SD card which is then inserted into the camera. The user then has to start the camera and request a firmware update in the main menu. Internally, the camera verifies the presence of an update file on the SD card, sets a flag in non-volatile memory and issues a system reboot.

Upon reboot, the stage-0 bootloader, executed from ROM, loads the stage-1 bootloader into RAM and transfers execution to it. The stage-1 bootloader then checks whether the `update_requested` flag is set and, if so, loads the update file from the SD card into RAM. Afterwards, the bootloader checks a field of the update file's header to ensure that the update targets the right camera model. If this is the case, a checksum is computed over the firmware file and compared with the corresponding field in the header.

If the checksum matches, the stage-1 bootloader copies another chunk of functions from ROM to RAM. These functions are responsible for decrypting the first part of the update file. After successful decryption, control is handed back to the stage-1 bootloader, which then in turn transfers execution to the freshly decrypted part of the firmware update file. This part, called the *flasher* by the Magic Lantern community[1], performs a variety of operations and eventually decrypts the rest of the firmware update.

[1] http://magiclantern.wikia.com/wiki/Packing_FIR_Files.

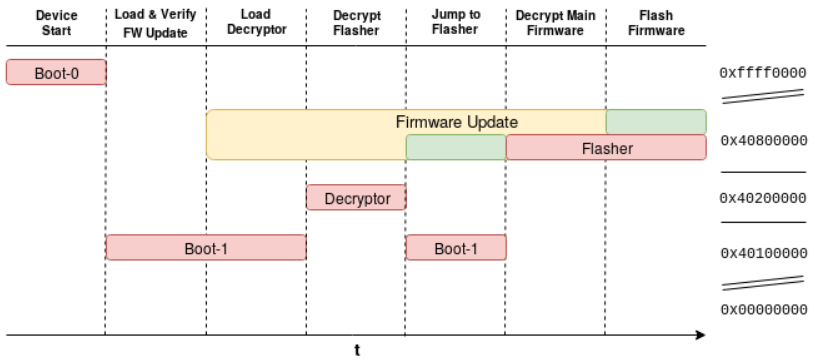


**Figure 6.2:** Canon EOS 60D update process.
Encrypted Data Decrypted Data Executed Code

### Groundhogger in Action

To carry out this case study, we first merged PANDA and the Groundhogger analysis passes into the Magic Lantern QEMU fork, which allows a complete emulation of the firmware code. To dump the firmware image running on our 60D, we installed the Magic Lantern add-ons, which saves the vanilla firmware of the camera to its SD card.

As this use-case reflects the perfect scenario by having access to full emulation, it is not necessary to use any heuristic to locate the entry point for the recording. Instead, we were able to record the entire execution of the firmware obtained by booting the device in "firmware update mode". The execution eventually hangs, probably due to the fact that not all peripherals are fully implemented by the Magic Lantern project.

Once the entire trace was collected, we fed it to Groundhogger, which automatically detected the input firmware location (i.e., the *input buffer*). It also tagged a total of 16 routines as functions of interest (i.e., those that operate on the input buffer). Interestingly, 11 of those routines were located inside the input buffer itself, hinting towards a multi-stage, nested update mechanism. This hypothesis was later confirmed by Groundhogger during its fifth analysis step, when it attempted to locate the output buffer and found that it was actually at the same position as the input buffer: the unpacking was performed in place by overwriting the data.

With the position of the two buffers and the list of functions of interest in hand, we had to identify the correct routines to extract the unpacker. Although in principle we could have simply tried all 16 candidates, Groundhogger was able to automatically pinpoint the two relevant functions by additionally marking those

which, according to the recorded trace, modified the content of the output buffer in addition to reading the input buffer.

This led to the successful identification of the unpacking routine that retrieved the flasher (which accounts for roughly 15% of the update file) as well as the second-stage routine inside the flasher itself. Note that while we successfully identified both decryption routines with Groundhogger, we did not need to analyze which kind of encryption was actually used to encrypt the firmware images.

Finally, since in this case Groundhogger extracted *two* independent unpacking routines, we decided to re-host them sequentially and independently of each other. Additionally, to prevent errors due to in-place modification of the firmware update file, we load the firmware to be unpacked into a new `update_file` buffer in an empty space in RAM. We then set the register holding the pointer to the buffer accordingly; both of these additional steps have been automated using `avatar`$^2$.

**Testing the Extracted Routines**

To test the extracted unpacking routines, we manually downloaded firmware updates for cameras of the EOS D product line available on Canon's website. Only 32 cameras in the EOS D product line (out of 56 models) had firmware updates available. Furthermore, for each model with an update, only the last update was available. The first unpacking routine was able to successfully unpack the flasher for 20 out of 32 firmware images. Then, by running the second extracted routine identified by Groundhogger, we could correctly unpack the full firmware image in all 20 cases.

Interestingly, firmware updates for Canon cameras with dual DIGIC cores contain two distinct flashers embedded in the firmware update file, one for each core. Although the 60D used as baseline for our unpacking efforts only comes with a single DIGIC core, Groundhogger was able to unpack both flashers for the other models without any problem.

In comparison, `fir_tool2`, a dedicated unpacking tool maintained by the Magic Lantern community, could only unpack the flasher for 9 images and was never able to unpack and retrieve the core firmware. The nine firmware images for which it was able to extract the flasher used a XOR-based decryption routine, while the images we unpacked with Groundhogger are for cameras released later than 2010 where the flasher is encrypted with AES.

In order to also unpack those 9 images, we obtained an additional firmware dump for a Canon 450D, one of the cameras whose update are supported by `fir_tool2`. We applied the Groundhogger analysis to it and could indeed unpack 9 flashers and

9 core images. The only drawback in these cases is that we were not able to retrieve the second flasher for dual DIGIC cores. We manually verified that the obtained unpacked firmware images were correctly unpacked using disassembly, inspecting strings and file entropy.

The remaining 3 firmware images which were neither unpacked by `fir_tool2` nor Groundhogger are for older devices (before 2007) and come with a different firmware update file format. We did not have a device to perform a record and to confirm if Groundhoggoer would be able to unpack those.

In summary, Groundhogger was able to unpack 29 out of 32 firmware images for the EOS D product line, without any complicated manual reverse engineering, while `fir_tool2`, which was based on manual reverse engineering by the Magic Lantern community was only able to partially unpack 9 out of 32. Note that our success at unpacking relies fully on the fact that Groundhogger successfully identified and extracted the two unpacking methods. This case study also demonstrates that Groundhogger can help minimize the amount of hardware needed in order to unpack as much firmware as possible: an analyst can obtain one device, determine which updates it can successfully unpack, obtain a second device among the remaining unpacked firmware images, and so on until all available firmware is unpacked.

### 6.4.2 Case II: Firmware Modification

#### Hardware Platform and Device Information

Our second case study demonstrates how Groundhogger can benefit the analyst even if unpacking is already possible with existing tools. We choose the FOSCAM FI8918W IP camera as target, which runs uCLinux and exposes both an UART and a JTAG interface on its PCB.

Firmware updates are available on the vendor's website for two different subsystems of the camera: the web UI and the base system. The update itself is initiated by uploading the firmware update file via the web interface of the camera.

#### Challenges

In this case, the firmware updates are neither signed nor encrypted, and they are completely unpackable by using binwalk. However, when we try to upload a modified firmware update to the web UI, it reports a checksum error. Hence, the motivation for this case-study is to identify the location of the checksum in the firmware update file and to re-host the checksum routine so that it can be used offline to prepare a modified firmware image (in our example we aim to modify the web UI update to steal the login credentials entered by the user).

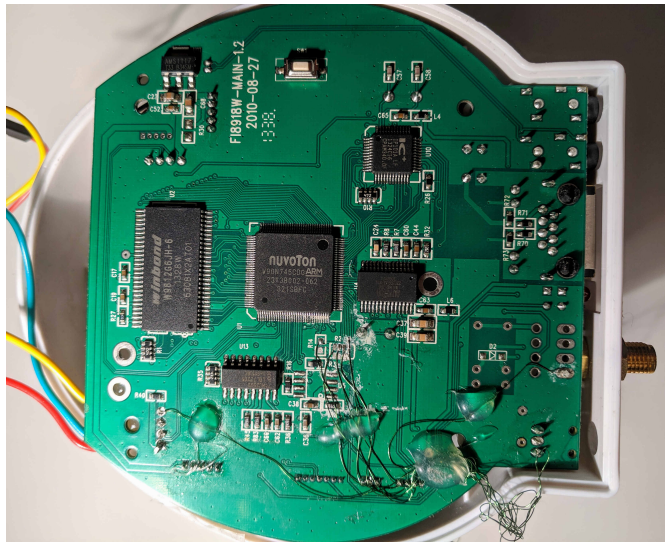


**Figure 6.3:** FOSCAM FI8918W prepared for Groundhogger.

The main challenge of this scenario is the fact that the device is running a complex operating system (uClinux). While emulation of a Linux-based firmware is generally easier, it presents an additional challenge to our analysis because the firmware update file is rarely present as a whole in a single input buffer in memory. Instead, it is accessed via the Linux syscall API and different parts are accessed and verified via `sys_open`, `sys_read` and `sys_llseek`.

### Groundhogger

For preparation, we have to attach to the JTAG interface of the camera's chip. Unfortunately, the interface is not exported to any header on the PCB. Nevertheless, it can be accessed via soldering directly on the chip packaging or attaching to the corresponding vias. We also soldered a connection to the UART pinout in order to obtain additional debug output. Our setup can be seen in Figure 6.3.

Finding a possible entry-point for the recording is relatively straightforward in this example. As the firmware update file is unpackable with binwalk, we have access to the full filesystem of the device, which includes a main control binary named `camera`. This binary is in ELF format and also provides a web server. As a result, identifying the `update_firmware` routine by manual reverse engineering is trivial. However, the functionality for retrieving the firmware over the network is not confined to a single method, but instead interleaved with other functions in the update procedure. As network traffic cannot easily be partially emulated, we followed a trial and error approach for finding a suitable entry point for the record.

In this scenario, the trace analysis required some changes to cope with the syscall-based nature of the Linux kernel. To deal with this, we adjusted the way Groundhogger identifies the input buffer and, instead of searching for the whole firmware update file, we instructed our tool to search for its filename.

During the initial replay, Groundhogger found the update filename in two distinct locations, one in the kernel and the other in user space. Using both of these as the input buffer, Groundhogger could identify a total of 11 functions of interest, of which 7 operate on the input buffer in kernel space.

We assumed that the update validation routine resides in user space and manually analyzed the remaining 4 functions of interest. By simply looking at the strings used in those functions, we could identify the function we want to rehost: one which combines `verify_metadata` and `verify_integrity`.

**Rehosting the Firmware Validation Procedure**

As the target function combines several tasks and opens, reads, and seeks the firmware update file *on the filesystem*, the sheer amount of resulting hardware accesses would challenge Groundhogger's rehosting capabilities under normal circumstances.

However, as the system is Linux-based, the hardware accesses are likely to be performed from kernel space. As a result, we can simply provide hooks for the necessary syscalls to rehost the routine[2]. To additionally simplify rehosting, we restrain execution of the function from its entry to the point of checksum validation.

In this period, the syscalls `sys_open`, `sys_read`, `sys_llseek`, `sys_ioctl`, and `sys_old_mmap` are executed. Interestingly, we only needed to provide hooks for the first three for successful rehosting.

Having the function rehosted, we can now use the hooks to operate on an input file of our choice, including a modified firmware update file. By introspecting the register contents during the checksum verification, we can automatically obtain the valid checksum value for an arbitrary update file and apply it. Note that, while this could also be done directly on the device, the advantage of Groundhogger is that we can now *export* the routine and execute it without the presence of a physical device.

This allows for easy firmware modification and can be used both offensively and defensively, e.g., for deploying patches or backdoors.

[2]Note that, while we provided our own hooks, reusable implementations of syscalls do exist, e.g., the simprocedures of the angr framework [SWS+16].

### 6.4.3 Case III: Unpacking and Modification

#### Hardware Platform and Device Information

Our third case study with Groundhogger targets the Crucial MX 100 SSD by Micron Technology. Previous studies reported that firmware updates are available on the vendor's website and that both UART and JTAG interfaces are present on the SSDs PCB, making it a perfect target for Groundhogger [CRB17, MvG19].

The vendor provides the firmware update in the form of a bootable ISO image, which installs the update automatically. To obtain the actual update file for the device, we extracted the content of the ISO image, which includes a live Linux system, the firmware update file, and an ELF executable responsible to upload the file to the SSD.

#### Challenges

The firmware update file itself appears to be neither compressed nor encrypted. Nevertheless, Binwalk is unable to extract any meaningful data from this image. When running the full update on the device, we can infer from debug information received over UART that the update is split into multiple segments which include, among others, a bootloader-code and an additional image for the MSP430 MCU. Additionally, four segments seem to be signed using RSA, which is in line with the findings of Meijer et al. [MvG19], who reported that the firmware is signed using RSA-2048 over SHA-256. The presence of cryptographic signatures make a firmware modification attack much more difficult. In fact, it is not possible to compute offline the required information as we did in the previous scenario. Instead, an attacker would need to tamper with the firmware already running in the disk to "trick it" into accepting the unsigned update.

Moreover, the type of device itself further complicates this scenario as the hard disk makes use of asynchronous I/O and DMA to transfer data. Partial emulation under these conditions is very difficult, requiring particular care to select a suitable entry point for the tracing procedure.

#### Groundhogger

This time, the initial step is to setup the physical device for its use with Groundhogger. We populated the JTAG and UART header reported by Cojocar et al. [CRB17] and connected the disk to a SATA to USB 3.0 adapter as shown in Figure 6.4. Once connected via JTAG, we stopped the execution during an ongoing update and dumped the RAM and Flash contents to obtain an initial snapshot of the firmware.

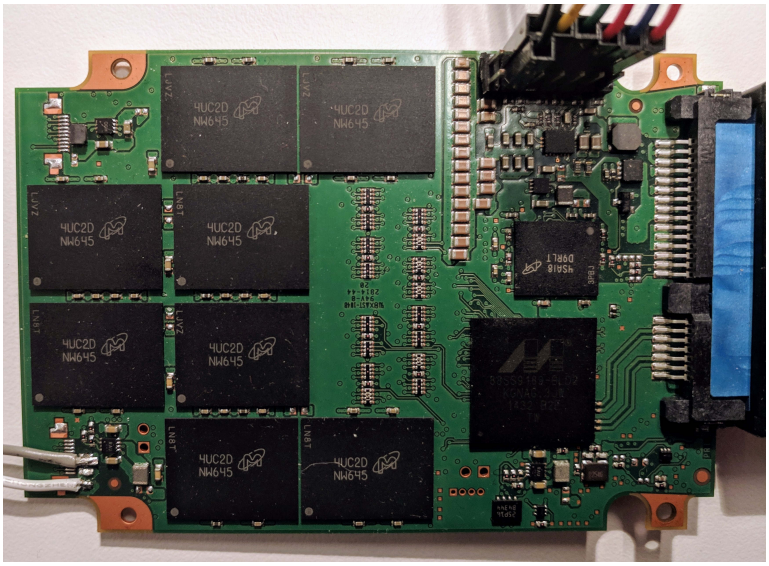

**Figure 6.4:** Crucial MX100 connected to Groundhogger.

We then located the entry point of the update procedure by disassembling the firmware image and cross-referencing the debug strings we observed during the update. However, in this case we needed several attempts to obtain an entry point suitable for partial emulation. In fact, the beginning of the update procedure heavily relies on hardware I/O accesses exceeding the capabilities of our partial emulation system. Therefore, we resorted to initiating the recording after the data was transferred and just before the firmware starts to verify the metadata.

Once the trace was collected, Groundhogger was able to easily identify the input buffer and tagged a total of 13 routines as functions of interest. These correspond to the routines responsible for verifying the headers, computing the CRCs, verifying the cryptographic signature of the firmware, and checking for magic numbers. While they all were good candidates for re-hosting, a single function unifying all these operations could not directly be found. Instead, we manually examined the functions *calling* the functions of interest in our record, and could easily identify a good target for re-hosting: a function which iterates over the segments in the firmware update file and verifies the checksum for each of those.

As this function combines a variety of different tasks and prints debug output, it contains hardware interactions which would normally not be re-hostable without additional analysis. However, thanks to avatar$^2$, we can easily provide hooks for critical functions. The efforts for doing so were minimal, as only two hooks needed to be provided to re-host the function. In more detail, we hooked `printf` to print

| Model | Version | #Segments |
|---|---|---|
| BX100 | MU02 | - |
| BX200 | MU02 | - |
| M550 | MU02 | 5 |
| MX100 | MU03 | 4 |
| MX100-128 | MU03 | 5 |
| MX200 | MU05 | 3 |
| MX300 | M0CR070 | 3 |
| MX300-2T | M0CR070 | 3 |
| MX500 | MU05 | - |

**Table 6.3:** Unpacking results for crucial SSDs.

the output on the analysis host, and skipped the execution of another function which checks the currently deployed firmware on the MSP430 MCU.

**Testing the extracted routine**

Once the function can be executed without the hardware, we used it to split the firmware image automatically into its different segments.

Afterwards, we downloaded the firmware updaters for other Crucial SSDs and retrieved nine other update files from the corresponding ISO images.

The re-hosted routine worked out of the box for other versions of the MX100 SSD firmware, but we had to slightly modify it to process updates for *different* models. The required change, not related to Groundhogger, was simply to prevent the validation routine (rehosted from an MX100 firmware) from discarding other firmware as not correct for the given device.

In total, we could extract the segments for 6 of the 9 update files, as shown in Table 6.3. The firmware updates which could not be unpacked were for SSDs of the BX series and for the MX500. Manual investigation showed that in all three of these cases the firmware packing format diverges from the one for the other disks.

**Enabling Firmware Modification Attacks**

Although the firmware is cryptographically signed, we can launch a modification attack against the connected SSD by using the JTAG debug port.

Groundhogger reported both check-summing and signature verification routines inside the set of the functions of interest. As a result, we know exactly, and with relatively low effort, which functions' return values have to be modified to deploy a rogue firmware update. In fact, we implemented a proof-of-concept attack in which we modified the contents in the segment for the main firmware. While we

only modified strings in the firmware, we want to point out that a more advanced version of the same attack could install a backdoor as reported in [ZBFB14]. To pass the firmware validation routines, we only needed to modify the return value of two functions, which we could identify easily due to Groundhogger.

# Chapter 7

# Outlook & Concluding Remarks

This chapter concludes this thesis. We discuss potential directions for follow-up and future work and wrap up with a brief conclusion.

## 7.1 Future Work

This thesis provides fundamental improvements to the art of dynamic binary firmware analysis. However, based on the insights won and challenges faced throughout this work, we want to give a perspective on unsolved problems and an outlook on future research directions.

First and foremost, omitting real vulnerabilities from our tests may raise the question whether the results of this work are applicable in a real world scenario. We believe this to be the case, as our work tackles binary firmware analysis on a conceptual level. Nonetheless, integrating the dynamic analysis techniques developed in this work in real world security tests will be highly interesting.

Furthermore, our contributions to dynamic binary firmware analysis are mainly based on rehosting the firmware under test. While rehosting as method to facilitate firmware analysis, it leaves space for exploration of other approaches, such as making use of hardware tracing capabilities—if available—or on-device firmware instrumentation.

In any case, rehosting itself provides a huge potential as research topic. Good emulators are the prerequisite for enabling powerful dynamic analysis in this case, and currently, they are manually hand-crafted for different hardware platforms. Hence, an important future research direction will be the *automated* generation of emulators, as it will allow for scalable dynamic binary firmware analysis. While

first steps in this direction are carried out in [GMS+19], more research in this direction is required.

Although fully automated rehosting remains the goal, partial emulation can be used as fallback strategy as long as good emulators are not available. Unfortunately, this requires the embedded device under analysis to expose debugging features by some means. While we mostly used hardware debugging features in this thesis, their availability on COTS devices can not be seen as granted. As a result, methods to gain easily a foothold on a device and versatile debugging stubs have to be developed. Furthermore, some hardware features, such as DMA, are difficult to partially emulate, and additional work is required to integrate them in partial emulation schemes.

Despite this additional research in the approach, the dynamic analysis techniques presented in this thesis leave room for future work. For instance, we purposefully omitted challenges in the realm of fuzzing which are not directly coupled to embedded systems. Other aspects of fuzzing, such as the generation of inputs to discover vulnerable paths, are likewise important for efficient testing and have to be subject of further studies.

Similarly, most symbolic execution engines are not ready for being used on firmware, and are hence often modified in a case-by-case manner. Having support for the particularities of embedded systems mainlined in popular engines would greatly benefit the future of dynamic binary firmware analysis.

Another question is how the availability of source code can aid the dynamic analysis of embedded systems. Sometimes, an analyst may have access to either the complete, or just parts, of the firmware source code. While this information can for sure improve the efficacy of dynamic firmware analysis, it is rarely used as *supportive* mean for binary analysis. Instead, tools and approaches making use of source code frequently assume the presence of the complete code, a build environment and toolchain.

Yet another potentially useful, but overlooked piece of information to aid dynamic firmware analysis are the abstractions given by the embedded operating systems of Type-II devices. While the majority of research on embedded systems which makes use of operating system abstractions solely focuses on Linux based devices, we believe that abstractions provided by other widespread operating systems, such as VxWorks, can be likewise useful.

Outside the realm of dynamic binary firmware analysis, we see one concept developed in this thesis especially promising: multi-target orchestration. With `avatar`$^2$, we created a potent tool for dynamic binary analysis in general, and its concepts

are adapted in the community, for instance to enable symbolic analysis of complex software [GGF18]. We see huge potential in using multi-target orchestration as building block for further research, as the possibility to interconnect and dynamically orchestrate different tools, debuggers and emulators is a powerful primitive.

## 7.2 Conclusion

In this thesis, we outlined and tackled the main problems for dynamic binary firmware analysis. In particular we identified five hindering factors limiting dynamic firmware analysis for security testing: (1) firmware retrieval, (2) platform variety, (3) fault detection, (4) scalability, and (5) instrumentation.

Through this work, we tackled each of these problems in an independent manner: We developed `avatar`$^2$, a multi-target orchestration framework designed to dynamically orchestrate a wide range of frameworks, debuggers, and emulators. This framework does not only allow emulation and instrumentation of embedded devices' firmware, but enables also partial emulation, which eases the challenge of platform variety.

Using the given instrumentation capabilities, we implemented a diverse set of run time fault detection heuristics which are mimicking existing techniques for fault detection commonly deployed on desktop systems.

Furthermore, we outlined the architecture for a scalable concolic execution testing framework, and showcased, Groundhogger, a novel approach for automating the difficult task of creating unpackers for embedded device firmware.

All in all, we believe that the work conducted in the thesis is a significant step toward better dynamic firmware analysis. Nevertheless, further research is still required to fully automate dynamic analysis of embedded system and to span a wider variety of devices.

# List of Tables

# List of Figures

# List of Acronyms

**JTAG** Joint Action Test Group

**AFL** American Fuzzing Lop

**COTS** commercial off-the-shelf

**CPU** Central Processing Unit

**DMA** Direct Memory Access

**EEPROM** electrically erasable programmable read-only memory

**ICS** Industrial Control Systems

**IRQ** Interrupt Request

**IR** Immediate Representation

**ISA** Instruction Set Architecture

**ISR** interrupt service routine

**MMIO** Memory-Mapped Input/Output

**MMU** Memory Management Unit

**MPU** Memory Protection Unit

**PLC** Programmable Logic Controller

**PMIO** Port-Mapped Input/Output

**ROM** read-only memory

**RTOS** Real-Time Operating System

**SCADA** supervisory control and data acquisition

**SNMP** Simple Network Management Protocol

**SoC** System-on-Chip

**SWD** Serial Wire Debug

**TCP** Transmission Control Protocol

**DSLR** digital single-lens reflex

**USART** Universal Synchronous/Asynchronous Receiver/Transmitter

**UART** Universal Asynchronous Receiver/Transmitter

# Bibliography


[ABSZ14] Magnus Almgren, Davide Balzarotti, Jan Stijohann, and Emmanuele Zambon. D5.3 report on automated vulnerability discovery techniques. *CRISALIS EU Project*, 2014.

[ALAM19] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[Alv] Sergi "pancake" Alvarez. Radare2: unix-like reverse engineering framework and commandline tools. http://radare.org. [Online; accessed 01-Aug-2019].

[AP07] Pedram Amini and Aaron Portnoy. Fuzzing Sucks! Introducing Sulley Fuzzing Framework. *Black Hat USA*, 2007.

[Art17] Nitay Artenstein. Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets. *Black Hat USA*, 2017.

[AVR14] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. Analysis of Embedded Applications By Evolutionary Fuzzing. In *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2014.

[BA04] Derek L Bruening and Saman Amarasinghe. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[BBLD13] Zachry Basnight, Jonathan Butts, Juan Lopez, and Thomas Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 2013.

[BCD+] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques.

[Bec13] Andre Beckus. QEMU with an STM32 microcontroller implementation. http://beckus.github.io/qemu_stm32/, 2013. [Online; accessed 02-Aug-2019].

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATC), FREENIX Track*, 2005.

[BGM13] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A Binary Analysis Platform. In *International Conference on Computer Aided Verification*. Springer, 2011.

[BLP+11] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation. *USENIX; login*, 2011.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[BZP19] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[Cap16] Pierre Capillon. Black-box cryptanalysis of home-made encryption algorithms: a practical case study. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2016.

[CCF18] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *USENIX Security Symposium*, 2018.

[CCM+18] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 2018.

[CCS13] Ang Cui, Michael Costello, and Salvatore Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[CDZ+18] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*. IEEE, 2018.

[CGC08] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference (ATC)*, 2008.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[CLW+18] 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 *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018.

[CN19] Christian Cadar and Martin Nowack. KLEE Symbolic Execution Tool Test-Comp 2019 Entry. https://test-comp.sosy-lab.org/2019/talks/19_KLEE.pdf, 2019. [Online; accessed 20-Jun-2019].

[CP16] Christian Collberg and Todd A. Proebsting. Repeatability in Computer Systems Research. *Communications of the ACM*, 2016.

[CRB17] Lucian Cojocar, Kaveh Razavi, and Herbert Bos. Off-the-shelf Embedded Devices as Platforms for Security Research. In *European Workshop on Systems Security (EuroSec)*. ACM, 2017.

[CS11] Ang Cui and Salvatore Stolfo. Defending Embedded Systems with Software Symbiotes. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer, 2011.

[CSB13] Xi Chen, Asia Slowinska, and Herbert Bos. Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries. In *Working Conference on Reverse Engineering (WCRE)*, 2013.

[Cui12] Ang Cui. Embedded Device Firmware Vulnerability Hunting Using FRAK. *Black Hat USA*, 2012.

[CWBE16] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[CZF16] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[CZFB14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.

[CZJ+15] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. Test Generation for Embedded Executables via Concolic Execution in a Real Environment. *IEEE Transactions on Reliability*, 2015.

[CZV+15] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. PIE: Parser Identification in Embedded Systems. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[CZZ+13] Ting Chen, Xiao-song Zhang, Cong Zhu, Xiao-li Ji, Shi-ze Guo, and Yue Wu. Design and implementation of a dynamic symbolic execution tool for windows executables. *Journal of Software: Evolution and Process*, 2013.

[DBXP20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. *To appear at IEEE Symposium on Security and Privacy (S&P)*, 2020.

[DCN+19] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2019.

[Del11] Guillaume DelugrÃl'. Reverse engineering a Qualcomm baseband. 28th Chaos Communication Congress (28C3), 2011.

[Des12] Fabrice Desclaux. Miasm : Framework de reverse engineering. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2012.

[DFPA17] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev.ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *International Conference on Compiler Construction*. ACM, 2017.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop (PPREW)*. ACM, 2015.

[DGHK+16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[DLRA15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding Integer Overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015.

[DMRJ13] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium*, 2013.

[DPY18] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018.

[Ebi16] Ebiroll. ESP32 in QEMU. https://github.com/ebiroll/qemu_esp32, 2016. [Online; accessed 02-Aug-2019].

[Fit18] Fitbit, Inc. Announcing Fitbit OS 2.0 and Our Brand New Simulator. https://dev.fitbit.com/blog/2018-03-13-announcing-fitbit-os-2.0-and-simulator/, 2018. [Online; accessed 15-Apr-2019].

[FPKS15] Ian D Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and Vulnerable: A Story of Telematic Failures. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[FZX+16] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[Gal17] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack, 2017. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[GBC+17] Luis Garcia, Ferdinand Brasser, Mehmet H Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A Zonouz. Hey, My Malware Knows Physics Attacking PLCs with Physical Model Aware Rootkit. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[GGF18] Fabio Gritti, Eric Gustafson, and Lorenzo Fontana. symbion: fusing concrete and symbolic execution. http://angr.io/blog/angr_symbion/, 2018. [Online; accessed 19-Jun-2019].

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[GLM+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 2012.

[GMS+19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Aurelien Francillon, Davide Balzarotti, Ryn Yung Choe, Christopher Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2019.

[GWH11] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 41–60. Springer, 2011.

[HC13] Craig Heffner and Jeremy Collake. Firmware Modification Kit. https://code.google.com/archive/p/firmware-mod-kit/, 2013. [Online; accessed 20-Jun-2019].

[Hea02] Steve Heath. *Embedded systems design*. Elsevier, 2002.

[Hef13] Craig Heffner. Binwalk: Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk, 2013. [Online; accessed 01-Aug-2019].

[Hem18] Armijn Hemel. Binary Analysis Next Generation (BANG). https://github.com/armijnhemel/binaryanalysis-ng, 2018. [Online; accessed 02-Aug-2019].

[Hem19] Armijn Hemel. Better unpacking binary files using contextual information. *Technical Disclosure Commons*, 2019.

[HFT+17] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[HJO18] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. REPICA: Rewriting Position Independent Code of ARM. *IEEE Access*, 2018.

[HKVD11] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding Software License Violations Through Binary Code Clone Detection. In *Working Conference on Mining Software Repositories (MSR)*. ACM, 2011.

[Hot16] George Hotz. Timeless debugging. San Francisco, CA, 2016. USENIX Association.

[HPY+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-System Dynamic Binary Analysis Platform. In *International Symposium on Software Testing and Analysis*. ACM, 2014.

[Jam13] Reinders James. Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013. [Online; accessed 02-Aug-2019].

[JCG+14] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis. In *Program Protection and Reverse Engineering Workshop (PPREW)*. ACM, 2014.

[KBK16] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[Kee16] Keen Security Lab. Car Hacking Research: Remote Attack Tesla Motors. http://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/, 2016. [Online; accessed 02-Aug-2019].

[Kei19] Keil. List of ARM-based Devices Available. http://www.keil.com/dd/chips/all/arm.htm, 2019. [Online; accessed 20-Jul-2019].

[Kel11] Stephen Kell. Static versus dynamic analysis—an illusory distinction? https://www.cs.kent.ac.uk/people/staff/srk21/

blog/research/static-and-dynamic-analyses.html, 2011. [Online; accessed 01-Aug-2019].

[Kin76] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[KKC+17] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017.

[KKJ12] Moonzoo Kim, Yunho Kim, and Yoonkyu Jang. Industrial Application of Concolic Testing on Embedded Software: Case Studies. In *International Conference on Software Testing, Verification and Validation*. IEEE, 2012.

[KKM15] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[KL13] Nassima Kamel and Jean-Louis Lanet. Analysis of HTTP Protocol Implementation in Smart Card Embedded Web Server. *International Journal of Information and Network Security*, 2013.

[KPK14] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. PROSPECT: Peripheral Proxying Supported Embedded Code Testing. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2014.

[LCC+15] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing CAN Packets into Automobiles. In *International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2015.

[LCM+05] 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. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[LFW+18] Qiang Li, Xuan Feng, Raining Wang, Zhi Li, and Limin Sun. Towards Fine-grained Fingerprinting of Firmware in Online Embedded Devices. In *IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[LN17] Ulf Lindqvist and Peter G Neumann. The Future of the Internet of Things. *Communications of the ACM*, 2017.

[LZL+16] Muqing Liu, Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu. Security Analysis of Vendor Customized Code in Firmware of Embedded Device. In *International Conference on Security and Privacy in Communication Systems (SecureComm)*. Springer, 2016.

[Man13] Felipe A Manzano. PySymEmu: An intel 64 symbolic emulator. https://github.com/feliam/pysymemu, 2013. [Online; accessed 02-Aug-2019].

[Max17] Clive Maxfield. Embedded Markets Study - Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments. Technical report, AspenCore, 2017.

[McM15] David McMillen. Security attacks on industrial control systems. how technology advances create risks for industrial organizations. Technical report, IBM Security, 2015.

[MEMS14] Jacob Maskiewicz, Benjamin Ellis, James Mouradian, and Hovav Shacham. Mouse Trap: Exploiting Firmware Updates in USB Peripherals. In *Workshop on Offensive Technologies (WOOT)*, 2014.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 1990.

[MGS11] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *USENIX Security Symposium*, 2011.

[MHH+18] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *arXiv preprint arXiv:1812.00140*, 2018.

[MMH+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg.

Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *arXiv preprint arXiv:1907.03890*, 2019.

[MNFB18] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *International Conference on Software Engineering (ICSE)*. IEEE, 2007.

[MSK+18] 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 *Network and Distributed System Security Symposium (NDSS)*, 2018.

[MV15] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *Black Hat USA*, 2015.

[MvG19] Carlo Meijer and Bernard van Gastel. Self-encrypting deception: weaknesses in the encryption of solid state drives. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[Net04] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.

[Nor16] Amy Nordrum. The Internet of Fewer Things. *IEEE Spectrum*, 2016.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[OJF+17] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record And Replay For Deployability. In *USENIX Annual Technical Conference (ATC)*, 2017.

[Pak12] Brian S Pak. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. Master's thesis, Carnegie Mellon University, 2012.

[Pea17] Peachtech. Peach Fuzzer Platform Whitepaper, 2017.

[Per16] Joshua Pereyda. boofuzz: A fork and successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz, 2016. [Online; accessed 01-Aug-2019].

[PGC18] Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its BMC: the HPE iLO4 case. In *Symposium sur la sécurité des technologies de lâĂŹinformation et des communications (SSTIC)*, 2018.

[QV15] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn - The ultimate CPU emulator. https://www.unicorn-engine.org/, 2015. [Online; accessed 02-Aug-2019].

[Ral17] Ralf, Nico. Emulation and exploration of bcm wifi frame parsing using luaqemu. https://comsecuris.com/blog/posts/luaqemu_bcm_wifi/, 2017. [Online; accessed 15-Jul-2019].

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[SCA+18] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2018.

[Sch16] Maarten Schellevis. Getting access to your own Fitbit data. Bachelor's thesis, Radboud University, 2016.

[Ser15] Kostya Serebryany. Simple guided fuzzing for libraries using llvmâĂŹs new lib-fuzzer. http://blog.llvm.org/2015/04/fuzz-all-clangs.html, 2015. [Online; accessed 02-Sep-2019].

[Ser16] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. San Francisco, CA, 2016. USENIX Association.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[She17] Team Shellphish. Cyber Grand Shellphish. Phrack Papers, 2017.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*. ACM, 2009.

[SLR+19] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (S&P)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005.

[SMB+18] Omer Shwartz, Yael Mathov, Michael Bohadana, Yossi Oren, and Yuval Elovici. Reverse Engineering IoT Devices: Effective Techniques and Methods. *IEEE Internet of Things Journal*, 2018.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[SS15a] Florent Saudel and Jonatahn Salwan. Triton: Concolic execution framework. In *Symposium sur la sécurité des technologies de lâĂŹinformation et des communications (SSTIC)*, 2015.

[SS15b] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[SWH+15] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng,

Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[TDMK18] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[TGC17] Sam L Thomas, Flavio D Garcia, and Tom Chothia. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2017.

[Tra16a] Trail of Bits. A fuzzer and a symbolic executor walk into a cloud. https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems/, 2016. [Online; accessed 02-Aug-2019].

[Tra16b] Trail of Bits. GRR: High-throughput fuzzer and emulator of DECREE binaries. https://github.com/trailofbits/grr, 2016. [Online; accessed 02-Aug-2019].

[TTZ+18] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *USENIX Security Symposium*, 2018.

[VDBHT14] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. Security Testing of GSM Implementations. In *International Symposium on Engineering Secure Software and Systems*. Springer, 2014.

[VOC18] Sebastian Vasile, David Oswald, and Tom Chothia. Breaking all the Things — A Systematic Survey of Firmware Extraction Techniques for IoT devices. In *Smart Card Research and Advanced Application Conference (CARDIS)*. Springer, 2018.

[Vyu15] Dmitry Vyukov. Syzkaller. https://github.com/google/syzkaller.git, 2015. [Online; accessed 02-Aug-2019].

[Wei12] Ralf-Philipp Weinmann. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *Workshop on Offensive Technologies (WOOT)*, 2012.

[WSB+17] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[YLX+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[Zal14] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014. [Online; accessed 02-Aug-2019].

[ZBFB14] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[ZDY+19] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[ZKB+13] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik Olivier Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and Implications of a Stealth Hard-Drive Backdoor. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.