ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0043

EDITE - ED 130

**Doctorat ParisTech**

# T H È S E

**pour obtenir le grade de docteur délivré par**

## TELECOM ParisTech

**Spécialité « Informatique »**

*présentée et soutenue publiquement par*

**Jonas ZADDACH**

le 3 août 2015

# Développement de nouvelles techniques d'analyse binaire pour l'analyse de la sécurité des systèmes embarqués

Directeurs de thèse : **Davide BALZAROTTI**
**Aurélien FRANCILLON**

**Jury**

| | |
|---|---|
| **Engin KIRDA**, Professeur, Northeastern University à Boston | Rapporteur |
| **Jean-Louis LANET**, Professeur, Inria Rennes et Université de Limoges | Rapporteur |
| **Renaud PACALAIS**, Directeur d'Études, EURECOM | Examinateur |
| **Miguel SANTANA**, Chercheur, ST Microelectronics à Lyon | Examinateur |

**TELECOM ParisTech**
École de l'Institut Télécom - membre de ParisTech

T H È S E

# Development of novel dynamic binary analysis techniques for the security analysis of embedded devices

Thesis

**Jonas Zaddach**
jonas.zaddach@eurecom.fr

École Doctorale Informatique, Télécommunication et Électronique, Paris
ED 130

August 3rd, 2015

**Advisor:**
Prof. Dr. Davide Balzarotti
EURECOM, Sophia-Antipolis

**Co-Advisor:**
Prof. Dr. Aurélien Francillon
EURECOM, Sophia-Antipolis

**Reviewers:**
Prof. Dr. Jean-Louis Lanet,
Université de Limoges, France

Prof. Dr. Engin Kirda,
Northeastern University, USA

**Examiners:**
Prof. Dr. Renaud Pacalet,
Télécom ParisTech, France

Dr. Miguel Santana,
ST Microelectronics, France

# Acknowledgements

First and foremost I would like to thank my supervisors, Davide Balzarotti and Aurélien Francillon. Their door was always open for me when there were questions, need of a brainstorming session, ideas to shape, and trivial organizational questions. I am very grateful for their guidance in the (sometimes rough) academic waters, which led me to this thesis. I would also like to thank my reviewers for their constructive comments regarding this thesis.

Then of course there were and are my friends and colleagues from the S3 group: Tamarrino, Davidino, Andrei, Onur and Merve, Luca, Leyla and Jelena. Thank you for all the time we spent together encouraging each other in the PhD, wisdom exchanged over a coffee or a juggling session in the office, and the good time outside of the office.

But Eurecom would not be the great place it is if I mentioned only my research group: across all departments I had and have a lot of friends, who made my days worthwhile. There was always somebody for a lunch break, a coffee break, and inspiring discussions. And of course a thank you goes to the administration for their help in every situation.

During my four-month stay in Amsterdam I enjoyed very much working in the group of Herbert Bos. Thank you Lucian for taking me out and showing me around, and thanks to my colleagues and the international dinner there for the evenings in the residence, the pub and the help in topics I did not know much about then!

Any doubts or worries I had during the PhD I could share with my friends in Antibes. Thank you Hadrien and Martina for countless cooking evenings, the international dinner people for the good time we had together, the students from my master promo at Eurecom, and all the others enriching my time outside of Eurecom. Though they were not as close in terms of geographical distance, my German friends provided constant support over telephone, during my stays in Germany and during their frequent visits. Thank you, Max, Ivan, Flo and Ingrid, and all the others who supported me! And not to forget my gracious hosts in Turine: Thank you for the countless times you have housed me on my way to Germany or back, Dino, Olta, and Antonella!

Last but not least, I want to express my gratitude to my parents. They listened to all of my problems during the PhD, and did their best to cheer me up from the distance, and were understanding when I holed up in my room to work while being on holidays in Germany.

Thank you all! Vielen Dank! Merci beaucoup! Grazie mille! Gracias! Teşekkür ederim!

# Abstract

Embedded devices are relevant for all aspects of our lives, and their security is a growing concern. Therefore it is highly important to perform analysis of embedded software, even when the source code or hardware documentation is not available. However, research in this field is hindered by a lack of dedicated tools. Advanced dynamic analysis, one of the principal methods of security analysis, is difficult to apply due to a lack of hardware support. Emulation, the obvious alternative to analysis on hardware, requires an accurate model of the hardware platform, and is in practice infeasible without a significant amount of reverse engineering.

To address these issues, we present *Avatar*, a dynamic binary analysis framework for firmware of embedded devices. Instead of emulating peripherals, *Avatar* orchestrates the execution of the firmware in an emulator and the physical device, forwarding peripheral accesses when needed. We describe several techniques to improve the performance of this approach, e.g., by optimizing the distribution of code and data between the two environments. To show the versatility of our framework to perform different security analyses, we also present the experiments we conducted in three use cases.

In the second part of this document, we demonstrate *Avatar*'s reverse engineering capacities on a commercial off-the-shelf hard disk drive. Using this knowledge, we then developed a prototype rootkit capable of replacing arbitrary blocks of data while they are written to disk. We further show how an attacker can establish a communication channel with the implanted backdoor of a compromised disk. No code modification of the server containing the disk is performed. Through this channel, any data stored on disk can be exfiltrated, e.g., the password database.

Last, we extended *Avatar* with a peripheral identification system to address some of the challenges of whole-system analysis. *Avatar* works well for the analysis of isolated code parts, but exhibits shortcomings when time-critical code, or symbolic execution with access to physical peripherals is involved. Our system generates a fingerprint of peripheral device interactions and then suggests similar peripherals from a database of collected fingerprints. Hence, a platform description can be recovered, which then serves to instantiate a custom emulator.

# Abstract en framçais

Les systèmes embarqués sont pertinents dans tous les aspects de notre vie, et leur sécurité est une préoccupation croissante. En conséquence, l'analyse des logiciels embarqués est d'une grande importance, même quand le code source ou la documentation du matériel n'est pas disponible. Néanmoins, la recherche dans ce domaine est freiné par le manque d'outils dédiés. L'analyse dynamique avancée, une des méthodes principales d'analyse de sécurité, est difficile à appliquer à cause de l'assistance manquante du matériel. L'émulation, une alternative évidente à l'analyse sur le matériel, requiert un modèle exact de la plate-forme physique, et est peu pratique en réalité sans un grand effort de rétro-conception.

Pour répondre a ces manques, nous présentons *Avatar*, un système d'analyse dynamique binaire des logiciel de systèmes embarqués. Au lieu d'émuler des périphériques, *Avatar* orchestre l'exécution du logiciel embarqué dans un émulateur et le matériel physique en relayant des accès au périphériques lorsque nécessaire. Nous décrivons plusieurs techniques pour améliorer la performance de cette approche, par exemple, en optimisant la distribution du code et des données entre les deux environnements. Pour démontrer la capacité de notre système d'assister dans des scénarios d'analyse de sécurité différents, nous présentons en outre les expériences que nous avons menés dans trois cas.

Dans la deuxième partie de ce document, nous démontrons les capacités de rétro-conception dans le cas d'un disque dur du marché consommateur. Avec les informations obtenues, nous avons développé un root-kit prototype capable de remplacer des blocs de données arbitraires au moment où ils sont écrits au disque. En plus, nous démontrons comment un attaquant peut établir un canal de communication en exploitant une porte dérobée d'un logiciel embarqué d'un disque dur infecté. Aucune modification du code du serveur contenant le disque est entreprise. Par ce canal, toutes les données stockées sur le disque peuvent être exfiltrées, par exemple, la base de données contenant les mots de passe.

Finalement, nous avons supplémenté *Avatar* avec un système d'identification des périphériques pour répondre à quelques défis d'émulation d'un système entier. *Avatar* fonctionne très bien pour l'analyse des parties de code isolées, mais démontre des faiblesses quant au code ayant des contraintes de temps réel, ou quand des périphériques physiques sont accédées durant l'exécution symbolique. Notre système génère une empreinte digitale des interactions avec le

matériel périphérique, puis suggère des périphériques similaires à partir d'une base d'empreintes digitales préalablement recueillies. Par conséquent, une description de la plate-forme peut être récupérée, ce qui sert ensuite à créer un émulateur personnalisé.

# Contents

# Chapter 1

# Introduction

Embedded systems have become ever more pervasive throughout our lives. Industrial systems cannot be thought of without computerized control, cars contain tens of electronic control units and millions of lines of code, and even home automation is getting increasingly popular.

While a compromise of a personal computer or server can cause significant problems and financial losses, hacked embedded systems can have an even more severe impact. Embedded systems are typically deployed to monitor and control processes in the physical world, where they can cause real, physical harm to humans and equipment. A good example is the *Stuxnet* worm, which infected workstations in a uranium enrichment facility in Iran in 2010. By inserting malicious commands in the centrifuges' control program, the worm likely destroyed hundreds of centrifuges and delayed the Iranian nuclear program significantly [NFC11, Lan13].

Until recently, embedded systems had mostly been decoupled from the Internet. However, this is now changing and so-called smart embedded devices, which do not only process local sensory input, but also receive data from other systems, are on the rise. The power grid would not be able to handle huge sudden changes of electricity production, as they happen with solar power, without weather forecasts. Industrial processes are predicted to gain in efficiency with what is termed "Industry 4.0" or "Industrial Internet of Things (IIoT)" [PD15]. And even in our homes, heating, light and door locks will be interconnected in the "Internet of Things (IoT)".

The real number of attacks against embedded systems is difficult to estimate. Reports of spectacular hacks in the media, like Stuxnet or a maliciously induced meltdown of a blast furnace in a German steel mill [bsi15], cast a spotlight on individual events. But embedded systems' exploits, for example for credit card terminals [NB14], Programmable Logic Controllers (PLCs) used for control in factories [Ber11], and switchable power outlets for home automation [Dav14],

are frequently discovered and presented at hacker conferences. The National Institute of Standards and Technology (NIST) vulnerability database contained only 190 entries concerning embedded device software in 2010, while the number increased almost ten-fold to about 1700 in 2014.

All these points illustrate that interconnected embedded devices can cause physical harm when not secured properly. As a result, governments show a raising awareness for the security of embedded devices, especially when they are employed in "critical infrastructure", like electricity, fuel and water supply [nis14, bsi15, ftc15].

### A lack of security in embedded devices

Embedded devices currently do not exhibit the same resilience against attacks as, for example, Personal Computer based systems do. First, embedded devices are typically long-running systems. During their lifetime, attack techniques and threats evolve significantly, while the device's embedded software, called firmware, is seldom or never updated. For example, a lot of consumer devices connected to the Internet are running outdated Linux kernels with known vulnerabilities [CZFB14].

Second, interconnectivity is sometimes added on top of existing, secure designs, leading to unintended security issues. While the solution of extending older, existing interfaces is an easy way to connect to older devices still running in the same system, the example of Modbus illustrates the dangers of this approach. Modbus [spe12] is an industry bus that was designed to connect local equipment in a factory. In 1999, it was extended to be used over TCP, without adding any provisions for security or authentication. As a consequence, a scan of the routable IPv4 address space revealed more than 12,000 Modbus devices which are directly connected to the Internet and accessible to anybody [Lal15].

Third, embedded device manufacturers had not received much pressure from their clients to provide secure systems, and had put the main development efforts in features and safety instead. Consumer devices like a television, where a new feature is likely more influential for a buying decision than a security certificate, are an example.

Fourth and most important, development of embedded devices is driven by cost considerations and time-to-market. Hardware security features invariably will need more silicon surface for their implementation, which translates to higher costs. Software security mechanisms usually require more memory or CPU power, which in turn translates to a more expensive chip. Devices which are running on limited power, like battery-driven sensor network nodes, might therefore sacrifice security for less power consumption. Finally, a robust security design necessitates additional development effort. Especially for bulk and consumer products, where

a low price is an important factor in the buying decision, manufacturers are tempted to forgo security.

### Embedded device are attractive attack targets

Embedded devices constitute an attractive target for attackers. Due to a lack of software management, they rarely receive updates as timely as Personal Computer (PC) systems do [CZFB14]. In addition, especially older devices are frequently shipped with insecure default configurations (e.g., easy-to-guess default passwords), which are hardened only by few users afterwards. Even worse, once an embedded device has been infected with malware, it is very hard today to detect the infection – as long as no deterioration in the original functionality of the device is visible, hardly any user will suspect malicious activity. Even if a malware is found, little can be done for most embedded devices to remove it reliably.

The lack of up-to-date, resilient code coupled with the fact that embedded devices, unlike PCs, are always-on, make them an attractive target for cybercriminals. While the heterogeneity of platforms still poses challenges to malware writers, exploit kits ease the development of malicious code by providing platform identification scripts and abstractions. A wave of exploits and worms for home routers illustrate this point: either the firmware or its configuration are changed to inject ads into websites [Fra15], steal banking credentials [cym14], or become part of a botnet [GAZ15, Car13].

Industrial embedded systems, which typically are separated from the Internet by network segregation, nevertheless face the same challenges of software management and malware detection. Hence, the more likely attack scenario for industrial systems are Advanced Persistent Threats (APTs), as the penetration of a corporate network requires more dedicated skills and effort. In these attacks, high-profile attackers (e.g., nation-states or a terrorist organizations) focus their malicious efforts against a particular system (e.g., a power plant) [NFC11, Lan13].

### Testing and reverse engineering assistance for embedded devices is needed

In summary, the previous points highlight the need for tools and techniques to improve the security of embedded devices. Even when parts or the whole source code of a firmware is available, binary analysis might be the only viable option for a third party, since the toolchain to build the firmware might not be provided. Moreover, when the source code does not build to exactly the same binary, which can happen due to different compiler versions or build environment configurations, it is very hard to prove that the source code corresponds to the original binary firmware. Thus, it may be easier to simply analyze the binary

firmware in the first place. In general, binary firmware analysis plays a very important role in several scenarios.

First, after a security incident happened, a post-mortem analysis is required to understand the breach and improve security practices accordingly. This work is typically done by investigators who do not have access to a firmware's source code. Any automated analysis assistance could speed up the process significantly.

Second, manufacturers and clients sometimes need to perform penetration tests. In this case, a device is typically tested as a black box. Interfaces are tested with invalid or corner-case values for known protocols, but no knowledge about the firmware itself is used in the process. By leveraging approaches like *guided fuzzing*, where knowledge about the firmware is used in the process, penetration testing could be made more efficient.

Third, proper testing of embedded devices is especially important for integration vendors. When several embedded devices are integrated into a larger system, white-box testing becomes especially important, as in addition to flaws of an individual device, the whole system can exhibit race conditions and deadlocks, which might occur extremely rarely and only under very specific circumstances.

### Static and dynamic analysis tools need to be adapted

Binary analysis for PC programs has greatly advanced in the last decade. White-box fuzzing of application and operating system interfaces, taint analysis to track data flow through applications, and symbolic execution to automatically discover new test cases and increase code coverage are advanced dynamic program analysis techniques which are frequently used by security analysts. Unfortunately, these techniques cannot be applied to embedded systems.

One reason is the heterogeneity of system software in embedded systems. While some systems have a clear separation and well-defined interface between the operating system and the application, like Linux-based embedded systems, others run special-purpose or proprietary operating systems. Especially in low-end embedded devices, one may even find the application compiled together with an operating system library.

Moreover, depending on the embedded system's purpose, hardware security features may differ a lot. High-end ARM processors have several privilege levels, and even support for hardware virtualization, while low-end processors of the same family can merely distinguish between three privilege levels and do not support virtual memory.

Static analysis is rendered difficult by the numerous instruction sets employed in embedded device processors. ARM is the most prevalent instruction set [CZFB14], but MIPS, AVR, MSP430, and 8051 are also frequently used. Just developing

tools for one instruction set is not a viable solution if one wants to provide an analysis which applies to embedded devices in general.

Finally, dynamic analysis methods which are more advanced than debugging and tracing are hard to use on embedded systems' software. Without special-purpose hardware, more advanced techniques require an instrumented emulator that runs the firmware. However, functioning of software and hardware of an embedded system is intertwined, such that a firmware will not execute correctly if not all platform peripherals are emulated at the expected addresses. Thus, one first needs to perform static analysis to reverse-engineer the platform before being able to build a suitable emulator and perform advanced dynamic analysis.

## 1.1 Problem statement

This dissertation is centered around the problem of dynamic binary firmware analysis. Independent analysts such as certification laboratories, penetration testers, forensic analysts, integration vendors and researchers have a legitimate interest in analyzing binary firmware for security purposes. But binary firmware analysis can be very difficult: Emulators need to be tailored to each embedded device specifically, as no common hardware abstraction exists. Even when one has a way to debug firmware running on a device, applying modern dynamic analysis methods like symbolic execution is impossible, as support from the hardware platform would be required. Instead, lots and lots of time is spent backtracking from crashes and following data flow by hand. Even the operating system, libraries and the application in a code blob need to be identified for each firmware again and again. With an environment that allows data flow analysis, code path exploration and instrumentation, more of these tasks could be automated.

## 1.2 Contributions

We propose an analysis framework which fills this gap of dynamic analysis tools for embedded devices. *Avatar*, as presented in Chapter 3, allows a user to emulate the firmware of an embedded device. Complex analysis applications such as concolic execution can be implemented on top of this framework. We discuss several techniques that can be used to optimize the performance of the system, and to adapt *Avatar* to the user's needs. *Avatar* is demonstrated in three different security scenarios, including reverse engineering, vulnerability discovery, and backdoor detection. To show the flexibility of our system, each test was performed on a completely different class of devices.

Further, in Chapter 4, we demonstrate a practical, real-world implementation of a data exfiltration backdoor for a common off-the-shelf SATA hard disk. On

this example, we show the dangers and catastrophic loss of security of malicious firmware modifications. The backdoor is self-contained, requiring no cooperation from the host. It is stealthy, in that it only hooks legitimate reads and writes, without relying on DMA or other advanced features. Its overhead is unnoticeable by the user in normal operation. This backdoor can be installed by software in very little time. We also demonstrate that it is feasible to build such a backdoor with an investment of roughly ten man-months, despite difficulties in debugging and reverse engineering a disk's firmware. Finally, we present a number of forensic techniques which can help to identify a similar backdoor.

Last, in Chapter 5 we propose a methodology to identify hardware configurations of embedded devices. *Avatar* tackles the problem of tight coupling between software and hardware by forwarding hardware accesses from the emulator where the software is running to the real hardware platform. This approach, while well-suited for the analysis of smaller code regions, has some disadvantages: Whole-system emulation is very difficult, as timing-critical code needs to be identified, interrupts need to be handled correctly, etc. Moreover, the forwarding of concrete values to physical hardware in symbolic execution renders all other symbolic states invalid. In this work, we observe the communication between firmware and peripherals in $S^2E$ to create a fingerprint of each peripheral device, similar to register model descriptions found in a human-readable datasheet. We show that a database of register models can be built, from which an automatic recommendation for a platform device map can be given.

## 1.3 Organization of the dissertation

In this dissertation, we analyze the feasibility and impact of a targeted attack on an embedded system, namely a Hard Disk Drive (HDD), and develop dynamic binary analysis tools to analyze such threats. After summarizing the state of the art in Chapter 2, we present *Avatar*, a dynamic analysis tool for embedded systems, in Chapter 3. In the next chapter, the design and implementation of a HDD firmware modification attack is shown. Chapter 5 then presents techniques based on *Avatar* to automate platform reverse engineering of embedded devices. Finally, the dissertation concludes in Chapter 6.

### Chapter 2 – Literature review

Chapter 2 summarizes the relevant state of the art. An overview of static, dynamic and symbolic binary analysis is given, which is relevant for all of the following chapters. The section on firmware security is mostly relevant for Chapters 3 and 4. The summary on backdoors is related to the work presented in Chapter 4, and finally driver and device reverse engineering is connected to Chapter 5.

### Chapter 3 – *Avatar*: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares

In this chapter, we present Avatar, a dynamic binary analysis framework for embedded devices. The introduction gives an overview of the design of embedded systems and the challenges in embedded device emulation. Following, the core idea of *Avatar*, the forwarding of Input-Output (I/O) accesses from an emulator to the physical device, is explained. Several techniques to improve the system's performance are discussed, and demonstrated on three different use cases. The publication that this chapter is based on has been published in the proceedings of the Network and Distributed Systems Symposium (NDSS) in 2014 [ZBFB14].

### Chapter 4 – Implementation and Implications of a Stealth Hard-Drive Backdoor

Here, we present a data exfiltration backdoor implanted in the firmware of a hard drive disk. The introduction motivates why a hard drive is an attractive attack target, and lays out the assumed threat model. Next, the reverse engineering of the disk's firmware and the implementation of the backdoor are discussed. Data exfiltration without assistance of the computer's operating system is outlined, and the backdoor's performance is evaluated. The work of this chapter has been presented at the 29th Annual Computer Security Applications Conference (ACSAC) in 2013 [ZKB+13].

### Chapter 5 – Towards automating platform reverse engineering of embedded devices

This chapter proposes a technique to reverse engineer the hardware platform of an embedded system. First, an overview of the challenges of whole-system analysis with *Avatar* are given. Then, a technique for fingerprinting accesses of the firmware to a peripheral device is described. Finally, limitations and future extensions of the method are discussed.

### Chapter 6 – Future work and conclusion

Finally, we summarize open research problems of each chapter, and conclude the dissertation with a review of the contributions of the previous chapters.

# Chapter 2

# Literature review

In this chapter, we summarize previous work related to this dissertation. Three sections are dedicated to static, dynamic and symbolic analysis of binary code. The relationship of these analyses is shown in Figure 2.1. Afterwards, one section for each following chapter describes embedded devices' firmware security (relevant for Chapter 3 and 4), backdoors (relevant for Chapter 4), and driver and device reverse engineering (relevant for Chapter 5).

## 2.1 Static binary analysis

Static analysis encompasses the extraction of knowledge from code without actually executing the code. There are still many open research problems in static code analysis: identifying and correctly disassembling machine instructions in the first place, reconstructing the control flow graph (especially in the presence of indirect jumps), reconstructing types (especially buffer bounds), and the extraction of models which describe the behavior of the code at a higher level. Binary code is very hard to analyze in its original form, which is why we studied several intermediate languages for code analysis and their ecosystems.

Because the focus of this dissertation lies on dynamic analysis, we use static methods mainly to analyze data flow inside basic blocks from traces of a dy-

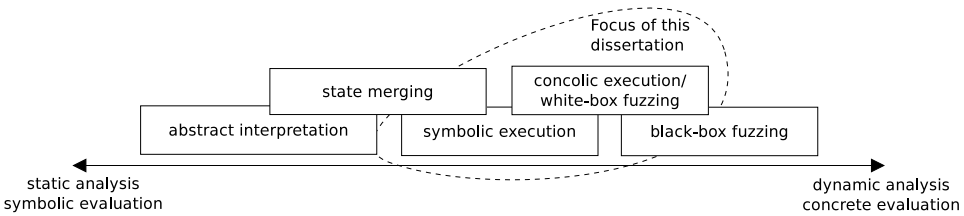


**Figure 2.1:** Relationship of static and dynamic binary analysis methods.

namic execution (in Chapter 5.3.2), and to statically identify control flow of code snippets to migrate to the embedded device (in Chapter 3.4.2). All of these static analyses were done using LLVM as intermediate language, which was chosen mainly for its good integration with the rest of the toolchain. Moreover, at the time when we had to decide for a framework, most of the alternatives had only incomplete or no support for the ARM instruction set.

### Intermediate languages

Machine instruction sets are inherently difficult to analyze, as many instructions have side effects which are not explicitly exposed. For example, the `MOV R0, R1` ARM assembler instruction, which copies the values of register `R1` to register `R0`, also implicitly modifies the program status register. A complete description with side effects would look like this:

```
R0 := R1
CPSR_N := R1 >> 31
CPSR_Z := R1 = 0
CPSR_C := 0
```

This is why machine programs, as originally proposed by [CS98], are usually translated to an intermediate language before analysis. Figure 2.2 shows how these languages relate to the compilation and reverse engineering process. Several frameworks have been built around these languages to assist reverse engineering. *Binary Analysis Platform (BAP)* [BJAS11], the successor of *BitBlaze* [SBY+08], translates to Bil. Bil is designed especially for reverse engineering and binary analysis and is backed by a formal definition. Once a program is translated to Bil, *BAP* provides an arsenal of tools for common static analyses.

*Valgrind* is another framework, originally designed for dynamic binary program instrumentation. Most known for its memory allocation checker, it has been extended by many other tools. *Pathgrind* [Sha14], for example, is a symbolic execution engine which can execute Vex, *Valgrind*'s intermediate language.

Yet another suite of (commercial) binary analysis tools, *BinDiff* and *BinNavi*, uses the Reverse Engineering Intermediate Language (REIL) for static machine code comparison and reverse engineering automation. The language has since been re-implemented in several frameworks, for example *BARF* [HA14], and in a modified form as RREIL in *bindead* [Mih].

*Radare2* [AiC] is a binary reverse engineering framework with support for static and dynamic analysis. Its intermediate language, Evaluable Strings Intermediate Language (ESIL), serves for interpreting code snippets on an analysis Virtual Machine (VM).

Finally, several frameworks exist to translate binary code to LLVM Intermediate Representation (IR) [LA04]. Since this intermediate language was not designed

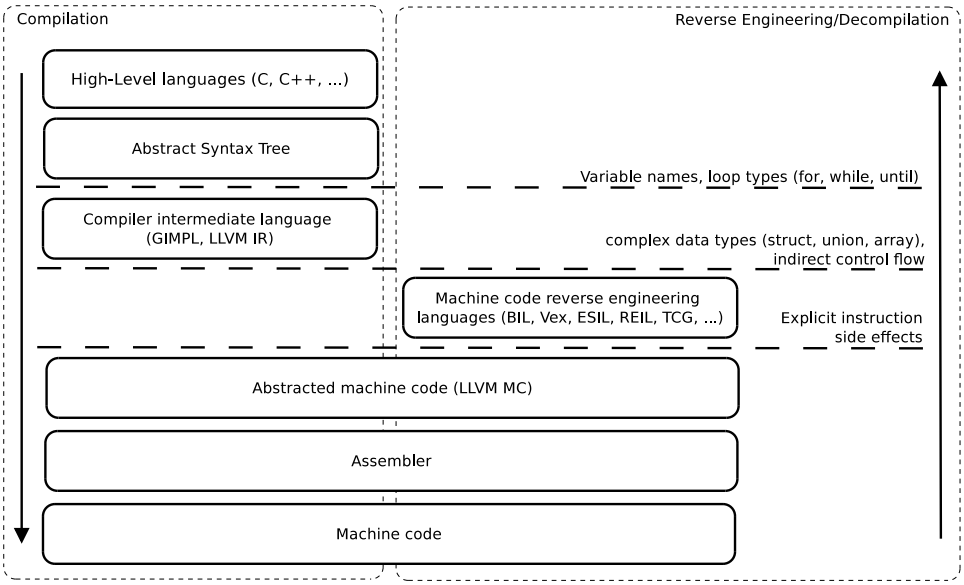


**Figure 2.2:** Positioning of intermediate languages in the compilation/reverse engineering process.

for reverse engineering, several constructs present in machine code are missing. For example, LLVM IR does not allow jumps to arbitrary destinations, but only to landing pads. Indirect jumps in machine code are difficult to impossible to resolve, which means that helper constructs like an "unknown" jump destination node need to be used if a jump is not definitely resolvable. Further, LLVM IR still uses functions, but most machine instruction sets do not have a unique way to express this abstraction. Thus, pattern-matching solutions to identify likely functions in machine code are used. Moreover, register assignments in LLVM use the Static Single Assignment (SSA) form, which is why the frameworks express processor state as global variables. Different static analysis frameworks (*McSema* [webc, DR14], *Dagger* [weba], *Fracture* [webb], *SecondWrite* [ASKE10], *RevGen* [CC11]) present solutions for these problems. *RevGen* is based on Qemu's translation of machine code to TCG, and *LLVM-Qemu* [CC10a], an additional translation layer from TCG to LLVM.

## 2.2 Dynamic analysis

Dynamic analysis is the technique of analyzing code while it is executing. Of these, trace recording and replay, as well as dynamic binary instrumentation are used heavily in this dissertation. More precisely, Chapter 5 requires recorded instruction and memory access traces, and binary instrumentation is a key component of Chapter 3.

## Tracing and replay

Program tracing records events during an execution. The range of possible events includes assembler instruction execution, basic block execution, conditional jumps, memory accesses, and many more. Depending on traced events, the taken execution path and even the register values during execution can be reconstructed post-mortem. Optimizations have been proposed [BCdJ+06] to minimize the impact of tracing.

A deterministic program can, once all its external inputs were recorded in a trace, be replayed. More precisely, the program will execute exactly the same code path when replayed, which allows for heavy instrumentation while the execution behavior remains deterministic. Replaying executions is difficult in parallel and distributed systems. Many solutions have been proposed for PC systems [XBH03, HA10, LSG+10], and some for embedded systems [DGHH+14].

Dedicated hardware support can provide a very good solution to improve efficiency of debugging, improving significantly the ability to replay events and system status. In [XBH03] Xu et al., presents an hardware architecture for recording precise events and replay them during debugging sessions. For this purpose custom hardware logs memory and taps on several important internal features (e.g., cache lines). Simpler systems also exist, like In-Circuit Emulators [Wil12], which replace the CPU core by an emulated CPU that can then directly interact with hardware peripherals.

*PANDA* [DGHH+14] is a dynamic analysis framework for whole system analysis. Instead of performing analyses directly when the program is running, it records execution traces, and then replays those traces with (possibly heavyweight) instrumentation. The benefit of this solution is that analyses are repeatable once traces have been recorded. *PANDA* has been used to identify interesting tap points in executables by monitoring data stored to and loaded from memory [DLHL13].

Similar to *PANDA*, we used *Avatar*, the framework presented in Chapter 3, to record and replay executions of embedded systems. We found replay to be an extremely valuable feature for advanced analyses such as concolic execution, data flow analysis and heavy instrumentation.

## Instrumentation

Static Binary Instrumentation (SBI) and Dynamic Binary Instrumentation (DBI) have both been used for some time on programs. DBI parses machine code before it is executed, and inserts instrumentation code on the fly (similar to a just-in-time compiling virtual machine). SBI inserts instrumentation hooks into the executable ahead-of-time, but can be used only when all code is known before

– dynamically loaded or generated code needs to be instrumented dynamically. DBI has been implemented as part of several frameworks.

*Valgrind* [NS07] is a heavyweight binary instrumentation framework, providing rich information about the program state at each point. *PIN* [RSCC04] is a proprietary instrumentation framework from Intel, which has also been ported to the ARM platform [HK06]. *DynamoRIO* [BDA01] has been recently used to instrument the Linux kernel [FBG12]. *DynInst* [BM11] offers both static and dynamic binary instrumentation.

Bochs [Law96] and the Qemu system emulator [Bel05] have both been instrumented for dynamic binary analysis [zyn, VE14, Lin12]. $S^2E$, which builds on Qemu, allows for whole-system dynamic binary instrumentation at instruction level through its plugin interface.

SBI is implemented in *PEBIL* [LTCS10], a framework which is supposed to provide more efficient instrumentation than dynamic instrumentation frameworks. An example of static binary instrumentation are $software\ symbiotes$, developed by Cui et al. [CS11], to insert calls to a security hypervisor into legacy firmware.

## 2.3 Symbolic execution

Symbolic execution as a program analysis technique has been proposed first by King et al. [Kin76]. Instead of executing a single program path conditioned by concrete inputs, symbolic inputs are provided to the program. Operations on symbolic values are tracked, and constraints are applied to symbolic values which influence program flow decisions. When a terminal program state is reached, the symbolic execution engine attempts to solve the generated equations in order to provide concrete inputs which will result in the execution of precisely the same execution path. In this way, concrete inputs for any possible execution path are generated.

A limitation of symbolic execution is *path explosion*. For each control flow decision in the executed path, at least two symbolic states are spawned. In particular counting loops and polling loops contribute to path explosion, as each loop iteration typically leads to two symbolic states.

A way to mitigate this problem is to use search heuristics which pick symbolic states for execution that are more likely to execute interesting paths [CDE08a]. For example, a heuristics which penalizes re-execution of already executed basic blocks helps to avoid getting stuck in loops and achieving greater code coverage. A second problem of symbolic execution is excessive memory consumption, as every state occupies some memory. State pruning is a way to reduce memory needs. Here, symbolic states which are very unlikely to yield a desired result are simply discarded [Cop14]. Of course both solutions require a deeper understanding of which paths are deemed interesting and which are not before the end of the

path is reached. Another way to reduce the number of symbolic states is state merging. Here, two states with almost identical path conditions are merged into one state with path conditions describing both previous states. As more path conditions stress the constraint solver, it is a good idea to only merge states when it is beneficial to the symbolic executor's overall performance [KKBC12]. When state merging is applied aggressively, symbolic execution converges towards abstract interpretation, a static program analysis technique.

Concolic execution [GKS05, SMA05, Sen07], also called "offline symbolic execution", reduces the resources needed by the symbolic execution engine by pairing a concrete and a symbolic input. A single path is executed, driven by the concrete value. At the end of the execution, one of the *path constraints*, deciding which execution path is chosen, is negated. By solving the new constraints, another concrete value driving the execution of this different program path can be found. Godefroid et al. use this technique in SAGE [GLM12] to create a white-box fuzzer for Windows applications. By permutating, for example, input files of applications, SAGE can generate example files which crash the program.

Many different symbolic execution engines for different programming languages and instructions sets have been built. Proof-of-concept systems [BEL75] existed quite early, but were limited by the computing power available at the time. Symbolic execution of full programs had been impractical due to the huge number of paths that need to be explored and the large amount of memory required to hold different states. Since then a large number of symbolic execution engines have sprung up. Some of them are only suitable for the execution of source code, like Otter for the C programming language [MYPFH11] and Rubyx for Ruby [CF10]. Others process intermediate languages for virtual machines like Java [VHB+03, LV01, SA06], .NET [TdH08], Dalvik [JMF12] and KLEE for LLVM [CDE08a].

While symbolic execution of machine code has long been difficult due to complex instruction sets and side effects of instructions, the rise of translators from machine code to intermediary languages has made this accomplishment possible [CS98]. Several different symbolic execution engines based on different intermediate languages have been presented. FuzzBALL [BMMS11, MMP+12] is based on the Vine intermediate language produced by the BitBlaze framework [SBY+08] which can execute x86 user space binaries. Similarly Pathgrind [Sha12] is executing x86 binaries translated to Valgrind's intermediate language. Mayhem [CARB12] and SAGE [GLM08] both have demonstrated that symbolic execution for vulnerability discovery is feasible on a large number of binaries.

$S^2E$ [CKC12] differs from the previously mentioned execution engines in that it can be applied to a whole platform, which enables symbolic execution of operating systems. This system couples the Qemu system emulator [Bel05] with KLEE. As long as no symbolic values are accessed, code is executed directly by

Qemu. Only when symbolic execution is required, a glue layer translates Qemu's internal intermediate language code, TCG, to LLVM, synchronizes the concrete and symbolic state, and defers execution to KLEE.

## 2.4 Embedded device firmware security

Embedded device security has often been answered with a "security by obscurity" approach by embedded systems' manufacturers, and has led to the discovery of major weaknesses in commonly deployed technologies [NESP08] in the past.

However, more rigorous solutions have been proposed based on virtualization, symbolic execution and binary instrumentation. Han et al. [HLSH11] propose a dynamic debugging system for Cisco IOS based on virtualization. Kuznetsov et al. [KCC10] present a testing system for binary device drivers in a virtualized environment using symbolic execution.

Similarly, Davidson et al. [DMJR13] developed a tool to perform symbolic execution of embedded firmware for MSP430-based devices. Like *Avatar*, this tool is based on the KLEE symbolic execution engine. However, it relies on the availability of firmware source code as well as on documented SoCs, peripherals mapping, and a simple device emulation layer. Any of those are rarely available for commercial devices.

In Firmalice [SWH+15], Shoshitaishvili et al. use a mixed approach of static, symbolic and manual analysis to identify authentication bypass backdoors in firmware. Points in the control flow which can only be reached when a user is authenticated are identified semi-automatically. The framework then assists the analyst in finding control flows which reach this point from an unauthenticated state without proper authentication (i.e., through hidden commands or hard coded credentials).

Cui et al. proposed *software symbiotes* [CS11], an on-device binary instrumentation to automatically insert hooks in embedded firmwares. Their solution allows to inject a security monitor that can interact with the original firmware.

## 2.5 Backdoors

Backdoors have a long history of creative implementations: Thompson [Tho84] describes how to write a compiler backdoor that would compile backdoors into other programs, such as the login program, and persist when compiling future compilers.

Many papers describe the design and implementation of hardware backdoors. King et al. [KTC+08] present the design and implementation of a malicious

processor with a circuit-level backdoor allowing, for example, a local attacker to bypass MMU memory protection. Heasman presents implementations of PCI and ACPI backdoors [Hea06, Hea07] that insert rootkits into the kernel at boot time. However, with the exception of Triulzi [Tri08], who presented a NIC backdoor that provides a shell running on the GPU, those previous backdoors were only *bootstrapped* from hardware devices. Then they tried to compromise the host machine's kernel from there. Therefore, those kinds of backdoors can be detected and prevented by kernel integrity protection mechanisms, such as Copilot [PJFMA04], which is implemented as a PCI device.

Cui et al. [CCS13] present a firmware modification attack on HP LaserJet printers. The authors remark that, in the case of most printers, firmware updates could be performed by sending specially-crafted printing jobs. Cui et al. also state that firmware updates were not signed and that signing would not prove sufficient in the presence of exploitable vulnerabilities, which is in line with our observations. In addition, they create, as payload, a VxWorks rootkit capable of print job exfiltration using the network link the printer is connected to.

Concurrently and independently from the work presented in Chapter 4, Domburg (a.k.a. sprite_tm) reverse-engineered a hard drive from another manufacturer and also demonstrated that modifying a hard disk firmware to insert a backdoor is feasible [aS], albeit without demonstrating data exfiltration.

Delugre [Del] reports on the techniques that were used to reverse engineer the firmware of a PCI network card, and to develop a backdoored firmware. For this purpose, QEMU was adapted to emulate the firmware and to forward IO access to the device. However, this was limited by bad performance. We have seen similar performance blockers when using *Avatar* in full separation mode, but the ability to perform memory optimization and push back code to the physical device allow *Avatar* to overcome such limitations.

Other examples of data-exfiltration attacks involving NICs include [SEZ09], where the authors use IOAPIC redirection to an unused IDT entry that they modify to perform data exfiltration. More generally, remote-DMA-capable NICs (such as InfiniBand and iWARP) can be used to perform data exfiltration [SB03]. However, such traffic can equally easily be identified and blocked by a firewall at the network boundary. Thus, a covert channel is needed to communicate with the backdoor, as mentioned in [Dae] for ICMP echo packets (independently of any hardware backdoor). In comparison, our approach leverages an existing channel on the backdoored system (e.g., HTTP) and therefore cannot be easily distinguished from legitimate traffic at the network level.

## 2.6 Driver and device reverse engineering

In this section, we lay out previous work on binary driver testing, reverse engineering and synthesis. The hardware reverse engineering presented in Chapter 5 is built upon similar techniques as used in driver reverse engineering, but aims at creating a model of a peripheral device, not its driver. For this reason, we also included works on driver testing with simulated devices, which is a close equivalent to our work. Finally, a subsection is dedicated to protocol learning. Especially the bit vector analysis technique used in the process of register data type recovery is very similar to the white-box analysis methods of Tupni and Prospex. Techniques for state machine recovery, as presented by Prospex, could be used in a future version of our peripheral reverse engineering system to build a more accurate fingerprint.

Several works for the reverse engineering of device drivers have been presented. RevNIC [CC10b] is a tool to automate reverse-engineering of device drivers. The authors demonstrate on the example of a Windows network driver that RevNIC can use symbolic execution to explore the device driver's code, slice instructions related to the driver, and build a synthesized driver from the extracted hardware model. SymDrive [RKS12] uses a very similar technique of exercising drivers with symbolic execution. The focus of this work is to find bugs in operating system drivers, without the need of the physical device that the driver is developed for.

Kuznetsov et al. [KCC10] analyze device drivers by relying on an emulated PCI bus and network card that return symbolic values. This approach has the main drawback that it requires to emulate the device properly. While this is not much of a problem for well understood devices, like a PCI network card supported by most PC emulation software, it can be a real challenge in embedded systems and can be just impossible when the hardware is not documented. Unfortunately, lack of documentation is the rule in the embedded world, especially in complex proprietary System on Chips (SoCs).

Guardrail [RKGM14] is a framework for run-time instruction-level driver analysis and can detect data races and uninitialized memory accesses in arbitrary kernel drivers. Levasseur et al. [LUSG04] propose a method for reusing unmodified device drivers running in isolated virtual machines. Faults in a driver thus only affect the virtualization domain, not the whole operating system.

Termite [RCK+09] reduces driver-related bugs by synthesizing drivers from formal specifications. In a way, the driver specification is exactly the other side of the coin of what we do in Chapter 5 – instead of building a specification of the driver, we extract a specification of the peripheral.

FEMU [LTHC10] proposes a hybrid firmware/hardware emulation framework for SoC. Thus, peripherals can be tested with emulated firmware. Several systems exist to simulate peripherals written in hardware specification languages

like VHDL or SysML for an emulated system running in Qemu [Zab12, Lem13, MHM12, SLC10].

### Protocol learning

Polyglot [CYLS07] differs from previous work on protocol reverse engineering in that it proposes a technique called shadowing to extract protocol specifications from a program binary. By observing how the program interprets received messages, the system is able to identify fixed length fields, variable length fields and keywords. The same approach of white-box execution analysis is followed by Tupni [CPC+08] to reverse binary file formats. In addition, it can use information from several example input files to gain more accurate information on file fields. Prospex [CWKK09] identifies similar protocol messages and clusters them to recover the protocol's state machine.

# Chapter 3

# Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares

*This chapter is based on a publication which has been presented at the Network and Distributed Systems Security Symposium (NDSS) in 2014 [ZBFB14].*

## 3.1 Introduction

An embedded system consists of a number of interdependent hardware and software components, often designed to interact with a specific environment (e.g., a car, a peacemaker, a television, or an industrial control system). Those components are often based on basic blocks, such as CPUs and bus controllers, which are integrated into a complete custom system. When produced in large quantities, such customization results in a considerable cost reduction. For large quantities, Application-Specific Integrated Circuit (ASIC) are preferred as they allow to tailor functionality according to the specific needs, which results in cost reduction, better integration, and a reduction of the total number of parts. Such chips, also called SoCs, are often built from a standard CPU core to which both standard and custom hardware blocks are added. Standard blocks, commonly called *IP Cores*, are often in the form of a single component that can be integrated into a more complex design (e.g., memory controllers or standard peripherals). On the other hand, custom hardware blocks are often developed for a specific purpose, device, and manufacturer. For example, a mobile phone modem may contain a custom voice processing Digital Signal Processor (DSP),

an accelerator for the Global System for Mobile Communications (GSM) proprietary hardware cryptography (A5 algorithms) and an off-the-shelf Universal Serial Bus (USB) controller.

Over the years, such SoCs have significantly grown in complexity. Nowadays, they often include Multiple Processors (MPSoC) and complex, custom, hardware devices. As a consequence, virtually every embedded system relies on a different, application specific, system configuration. As a witness of this phenomenon, the website of ARM Ltd., which provides one of the most common CPU core used in embedded systems, lists about 200 silicon partners[1]. Most of those partners are producing several product families of SoCs relying on ARM cores. This leads to a huge number of systems on the market, which are all different, but all rely on the same CPU core family.

Unfortunately, the increasing pervasiveness and connectivity of embedded devices significantly increased their exposure to attacks and misuses. Such systems are often designed without security in mind. Moreover visible features, low time to market, and reduction of costs are the common driving forces of their engineering teams. As a consequence, an increase in the number of reports of embedded systems exploitation has been recently observed, often with very serious consequences [BBB09, Car13, CMA+11, CCS13, Del, FMC11, MGS11, PD, Tri, ZKB+13]. To make things worse, such systems frequently play an important role in security-relevant scenarios: they are often part of safety critical systems, integrated in home networks, or they are responsible to handle personal user information. Therefore, it is very important to develop the tools and techniques that would make easier to analyze the security of embedded systems.

In the traditional IT world, dynamic analysis systems play a crucial role in many security activities - ranging from malware analysis and reverse engineering, to vulnerability discovery and incident handling. Unfortunately, there is not an equivalent in the embedded system world. If an attacker compromises the firmware of a device (e.g., a smart meter or a PLC in a Stuxnet-like attack scenario [FMC11]) even vendors often do not have the required tools to dynamically analyze the behavior of the malicious code.

Dynamic analysis allows users to overcome many limitations of static analysis (e.g., packed or obfuscated code) and to perform a wide range of more sophisticated examinations [ESKK08] - including taint propagation [KMPS11, WWGZ10], symbolic and concolic execution [CDE08b, CKC12, DMJR13], unpacking [KPY07], malware sandboxing [anu, CWS08], and whitebox fuzzing [GLM08, GLM12].

Unfortunately, all these techniques and their benefits are still not available in the world of embedded systems. The reason is that in the majority of the cases they require an emulator to execute the code and possibly monitor or alter its

[1] http://www.arm.com/community/partners/silicon.php

execution. However, as we will explain in Section 3.2, the large number of custom and proprietary hardware components make the task of building an accurate emulator a daunting process. If we then consider that additional modules and hardware plugins should be developed for each embedded system on the market, we can easily understand the infeasibility of this approach.

In this chapter, we present a technique to fill this gap and overcome the limitation of pure firmware emulation. Our tool, named *Avatar*, acts as an orchestration engine between the physical device and an external emulator. By injecting a special software proxy in the embedded device, *Avatar* can execute the firmware instructions inside the emulator while channeling the I/O operations to the physical hardware. Since it is infeasible to perfectly emulate an entire embedded system and it is currently impossible to perform advanced dynamic analysis by running code on the device itself, *Avatar* takes a hybrid approach. It leverages the real hardware to handle I/O operations, but extracts the firmware code from the embedded device and $emulates$ it on an external machine.

## 3.2 Dynamic Firmware Analysis

While the security analysis of firmwares of embedded devices is still a new and emerging field, several techniques have been proposed in the past to support the debugging and troubleshooting of embedded systems.

Hardware debugging features (mostly built around In-Circuit Emulators [CCK94, KHC08, Mel97] and JTAG-based hardware debuggers [JTA90]) are nowadays included in many embedded devices to simplify the debugging procedure. However, the analysis remains extremely challenging and often requires dedicated hardware and a profound knowledge of the system under test. Several debugging interfaces exist, like the Background Debug Mode (BDM) [Wil12] and the ARM CoreSight debug and trace technology [Wil12]. Architecture-independent standards for debugging embedded devices also exist, such as the IEEE NEXUS standard [IEE03]. Most of these technologies allow the user to access, copy, and manipulate the state of the memory and of the CPU core, to insert breakpoints, to single step through the code, and to collect instructions or data traces.

When available, hardware debugging interfaces can be used to perform certain types of dynamic analysis. However, they are often limited in their functionalities and do not allow the user to perform complex operations, such as taint propagation or symbolic execution. In fact, these advanced dynamic analysis techniques require an instruction set simulator to interpret the firmware of the embedded target. But for a proper emulation of the embedded system, not only the CPU, but all peripheral devices need to be emulated. Without such a support, the emulated firmware would often hang, crash, or in the best case, show a different behavior than on the real hardware. Such deviations can be due, for example, to

incorrect memory mappings, active polling on a value that should be changed by the hardware, or the lack of the proper hardware-generated interrupts or DMA operations.

To overcome these problems, researchers and engineers have resolved to three classes of solutions, each with its own limitations and drawbacks:

- *Complete Hardware Emulation*
  Chipounov [CC10b] and Kuznetsov et al. [KCC10] analyze device drivers by relying on an emulated PCI bus and network card that return symbolic values. This approach has the main drawback that it requires to emulate the device properly. While this is not much of a problem for well understood devices, like a PCI network card supported by most PC emulation software, it can be a real challenge in embedded systems and can be just impossible when the hardware is not documented. Unfortunately, lack of documentation is the rule in the embedded world, especially in complex proprietary SoCs.

  In some cases, accurate system emulators are developed as part of the product development to allow the firmware development team to develop software while the final hardware is still not available. However, those emulators are usually unavailable outside the development team and they are often not designed for code instrumentation, making them unable to perform basic security analysis like tainting or symbolic execution.

- *Hardware Over-Approximation*
  Another approach consists in using a generic, approximated, model of the hardware. For example, by assuming interrupts can happen at any time or that reading an IO port can return any value. This approach is easy to implement because it does not require a deep knowledge of the real hardware, but it can clearly lead to false positives, (e.g., values that will never be returned by the real system) or misbehavior of the emulated code (when a particular value is required). This approach is commonly used when analyzing small systems and programs that are typically limited to a few hundreds lines of code, as showed by Schlich [Sch10] and Davidson et al. [DMJR13]. However, on larger programs and on complex peripherals this approach will invariably lead to a state explosion that will prevent any useful analysis.

- *Firmware Adaptation*
  Another approach consists in adapting the firmware (or in extracting limited parts of its code) in order to emulate it in a generic emulator. While this is possible in some specific cases, for example with Linux-based embedded devices, this technique does not allow for an holistic analysis and may still be limited by the presence of custom peripherals. Moreover, this approach is not possible for monolithic firmwares that cannot be easily split into independent parts - unfortunately a very common case in low-end embedded systems [CS11].

In the next section we present our novel hybrid technique based on a combination of the actual hardware with a generic CPU emulator. Our approach allows to perform advanced dynamic analysis of embedded systems, even when very little information is available on their firmware and hardware, or when basic hardware debugging support is not available. This opens the possibility to analyze a large corpus of devices on which dynamic analysis was not possible before.

## 3.3 Avatar

*Avatar*[2] is an event-based arbitration framework that orchestrates the communication between an emulator and a target physical device.

*Avatar*'s goal is to enable complex dynamic analysis of embedded firmware in order to assist in a wide range of security-related activities including (but not limited to) reverse engineering, malware analysis, vulnerability discovery, vulnerability assessment, backtrace acquisition and root-cause analysis of known test cases.

### 3.3.1 System Architecture

The architecture of the system is summarized in Figure 3.1: the firmware code is executed inside a modified emulator, running on a traditional personal computer. Any IO access is then intercepted and forwarded to the physical device, while signals and interrupts are collected on the device and injected into the emulator.

The internal architecture is completely event-based, allowing user-defined plugins to tap into the data stream and even modify the data as it flows between the emulator and the target.

In the simplest case *Avatar* requires only a backend to talk to the emulator and one to talk to the target system, but more plugins can be added to automate, customize, and enhance the firmware analysis. In our prototype, we developed

[2] The *Avatar* framework is open-source and available at http://s3.eurecom.fr/tools/avatar.

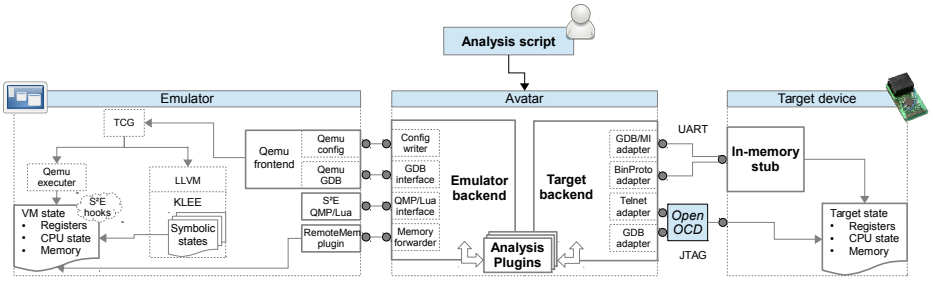


**Figure 3.1:** Overview of *Avatar*.

a single emulator backend. This controls S$^2$E (or Selective Symbolic Execution engine), which is an open-source platform for selective symbolic execution of binary code [CKC12]. It builds on the foundation of Qemu, a very popular open-source system emulator [Bel05]. Qemu supports many processor families such as i386, x86-64, Arm, Mips and many others. Apart from being a processor emulator, Qemu can also mimic the behavior of many hardware devices that are typically attached to the central processor, such as serial ports, network cards, displays, etc.

S$^2$E leverages the intermediate binary code representation of Qemu called Tiny Code Generator (TCG), and dynamically translates from TCG bytecode to Low-Level Virtual Machine (LLVM) bytecode whenever symbolic execution is active [LA04]. KLEE, the actual symbolic execution engine, is then taking care of exploring the different execution paths and keeps track of the path constraints for each symbolic value [CDE08b]. Evaluating possible states exhaustively, for some symbolic input, can be assimilated to model checking and can lead to proving some property about a piece of software [KKBC12].

Even though S$^2$E uses the TCG representation of the binary code to generate LLVM code, each processor architecture has its own intricacies that make it necessary to write architecture specific extensions to make S$^2$E work with a new processor architecture. Since our focus was on embedded systems and all the systems we analyzed are ARM systems, we updated and improved an existing incomplete ARM port[3] of S$^2$E, to suit the needs of dynamic analysis of firmware binaries.

To control the execution of code in more detail, S$^2$E provides a powerful plugin interface that allows instrumentation of virtually every aspect of execution. Any emulation event (e.g., translation of a basic block, instruction translation or execution, memory accesses, processor exceptions) can be intercepted by a plugin, which then can modify the execution state according to its needs. This modular architecture let us perform dynamic analysis of firmware behaviour, such as

[3] Our patches have been submitted to the official S$^2$E project and are currently under review for merging.

recording and sandboxing memory accesses, performing live migration of subroutines (see Section 3.3.3), symbolically executing specific portion of code as well as detecting vulnerabilities (see Section 3.5).

$S^2E$ is connected through three different control interfaces with *Avatar*: the first interface is a GDB debug connection using the GDB serial protocol. *Avatar* is connecting to this interface using a GDB instance controlled via the GDB/MI protocol. This connection is used for fine-grained control over the execution, such as putting breakpoints, single-stepping the execution, and inspecting register values. The second interface is Qemu's Management Protocol (QMP) interface, a JSON-based request-response protocol. Though detailed virtual machine control is possible through this interface, it is currently only used to dynamically change $S^2E$'s configuration at run time. This is done by accessing $S^2E$ through its Lua interface, which is called from Lua code embedded in the JSON requests. The third interface is a plugin for $S^2E$ that is triggered whenever a memory access is performed. This $S^2E$ plugin then forwards this request to *Avatar*, which in turn handles the memory access (e.g., sends it to *Avatar*'s plugins), or forwards it to the target.

Even though at the moment the only available emulator back-end is for Qemu/$S^2E$, the emulator interface is generic and allows other emulators to be added easily.

On the target side, we developed three back-ends:

- A back-end that uses the GDB serial protocol to communicate with GDB servers (e.g., a debugger stub installed on the device or a JTAG GDB server).
- A back-end to support low-level access to the OpenOCD's JTAG debugging interface via a telnet-like protocol.
- A back-end that talks to a custom *Avatar* debugger proxy over an optimized binary protocol (which is more efficient than the verbose protocol used by GDB). This proxy can be installed in an embedded device that lacks debugging hardware support (e.g., no hardware breakpoints) or on which such support was permanently deactivated.

The proper target back-end has to be selected by the user based on the characteristics and the debugging functionalities provided by the hardware of the embedded device. For example, in our experiments we used the OpenOCD back-end to connect to the JTAG debugger of the mobile phone and of the Econotag, while we used the *Avatar* proxy to perform dynamic analysis of the hard drive firmware.

To analyze a firmware, an access to the firmware's device is needed. This can be either a debugging link (e.g., JTAG), a way to load software or a code injection vulnerability. In cases where a debugging stub, for example the GDB stub, is used, an additional communication channel, e.g., an UART, is also needed.

### 3.3.2 Full-Separation Mode

When *Avatar* is first started on a previously unknown firmware, it can be run in what we call *"full-separation mode"*. In this configuration, the entire firmware code is executed in the emulator and the entire (memory) state is kept in the physical device. In other words, for each instruction that is executed by the emulator, the accessed memory addresses are fetched from and written to the real memory of the embedded system. At the same time, interrupts are intercepted by the debugging stub in the physical system and forwarded back to the emulator. Code and memory are perfectly separated, and *Avatar* is responsible to link them together.

Even though this technique is in theory capable of performing dynamic analysis on unknown firmwares, it has several practical limitations. First of all, the execution is very slow. Using a serial debug channel at 38400 Baud, the system can perform around five memory accesses per second, reducing the overall emulation speed to the order of tens instructions per second. Even worse, many physical devices have time-critical sections that need to be executed in a short amount of time or the execution would fail, making the system crash. For example, DRAM initialization, timer accuracy and stability checks belong to this category.

Moreover, tight hardware-polling loops (e.g., UART read-with-timeout) become painfully slow in full separation mode. Finally, regular interrupts (e.g., the clock tick) quickly overload the limited bandwidth between the target system and the emulator.

These limitations make the full separation approach viable only to analyze a limited number of instructions or when the user wants to focus only on particular events in more complex firmwares. For this reason, *Avatar* supports arbitrary context-switching between the emulator and the real device.

### 3.3.3 Context Switching

While it is possible to run the firmware code from beginning to end inside the emulator, sometimes it is more efficient to let the firmware run natively on the target device for a certain amount of time. This allows, for example, to execute the code without any delay until a particular point of interest is reached, skipping through initialization routines that may involve intensive I/O operations or network protocol communications that may need to be performed in real-time. In such cases, it is important to let the target device run the firmware, while still monitoring the execution for regions of code relevant to the current analysis. The ability of *Avatar* to perform arbitrary context switches gives the user the ability to quickly focus her analysis on a particular section of the code, without the drawbacks of emulating the entire firmware execution. In its core, this state migration technique is highly influenced by existing solutions for performance

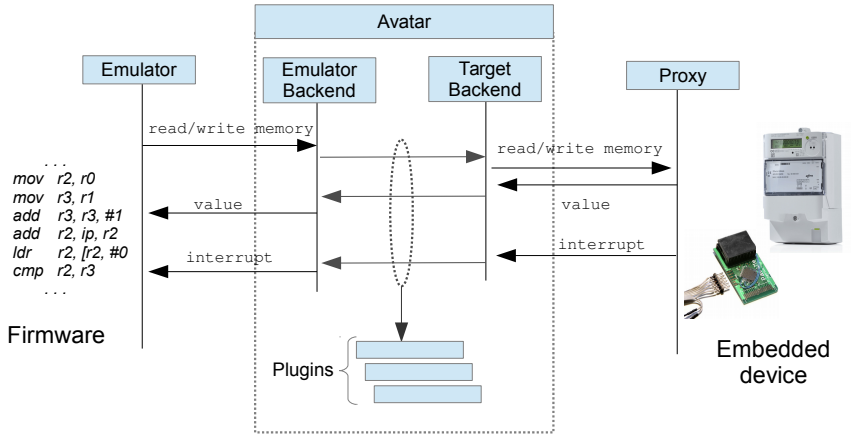


**Figure 3.2:** *Avatar* architecture and message exchange in full separation mode.

improvement of virtual machine hot-migration. In particular, our approach is a simplified version of the one proposed by Clark et al. [CFH+05], where *Avatar* is the arbiter of a *managed migration*, which can either happen in a single *stop-and-copy phase* (as in full-separation mode) or in an event-driven *pull-phase* (during context switching).

### Starting the analysis at specific points of interest

In this case the firmware starts the execution on the physical device and runs natively until a certain pre-defined event occurs (e.g., a breakpoint is reached or an exception is raised). At this point, the execution on the physical device is frozen and the state (e.g., the content of the CPU registers) is transferred to the emulator, where the execution is resumed. An example of this transition is described in Section 3.6.3, in which the firmware of a mobile phone baseband chip is executed until the phone receives an SMS, and then transferred by *Avatar* in the emulator to perform further analysis.

### Returning execution to the hardware

After the required analysis is performed on the emulator, the execution of the firmware can be transferred back to continue on the real device. In this case, any state kept on the virtual environment is copied back to the physical device. Depending on the user's needs, it is possible to switch again to the emulator at a later stage. This approach is used in Section 3.6.1, in which the firmware of a hard disk is started inside the emulator and later transferred back to the disk.

### 3.3.4 Interrupts Handling

Software interrupts do not present a problem for our framework, since they are issued by the firmware code and the emulator takes care of calling the corresponding interrupt handler directly. However, as shown in Figure 3.2, hardware interrupts need to be trapped in the real hardware and forwarded back to the emulator. In this case, the stub in the embedded system receive the interrupt and forwards them to *Avatar*'s target back-end. Finally, using the emulator back-end, *Avatar* suspends the firmware execution and injects the interrupt in the emulator.

Based on the circumstances in which the interrupt is generated, we distinguish three different cases:

- Hardware interrupts that indicate the completion of a task. These interrupts are issued by a device to indicate that a particular task initiated by the code has been completed. For example, the UART $send$ interrupt indicates that the send buffer has been successfully transmitted. This type of interrupts is easy to handle because it just needs to be forwarded from the target to the emulator.

- Periodical hardware interrupts, e.g., the timer notifications. These interrupts can be forwarded to the emulator but their frequency needs to be scaled down to the actual execution speed in the emulator. The equivalent number of instructions between two interrupts should be executed in the emulator as it would on the target running in native mode. In our current implementation, an *Avatar* plugin detects periodic interrupts and report their information to the user, who can decide how to handle each class. For example, the user can instruct *Avatar* to drop the clock interrupts on the device and just generate them (at the right frequency) on the emulator, thus saving bandwidth and increasing the analysis performance.

- Hardware interrupts that notify of an external event. For example the $receive$ interrupt of an UART indicates that new data on the UART buffer is available. The emulation strategy for those interrupts depends on the frequency of the external event. For events that require previous activity (e.g., a request-response protocol where the response triggers an interrupt) a simple forwarding strategy can be used. For unrelated events that happen very frequently (i.e., where the handler in the emulator cannot process the interrupt in time before the next interrupt is generated) the user can choose if she wants to suppress some of them or to handle the interrupt by migrating the handler itself back to the embedded device (see Section 3.4)

While the straightforward interrupt forwarding does not present any problem for *Avatar*, when the user needs to tune the framework to handle specific cases

(e.g., regular or very frequent interrupts) the stub needs to be able to distinguish between them. Unfortunately, this task is often difficult.

#### Interrupts de-multiplexing

In a traditional, x86-based, personal computer there is a standard interrupt controller that handles interrupt lines from each device and peripheral. However, on ARM-based systems there are only two interrupt lines directly attached and visible to the processor: `IRQ` and `FIQ`. Because of this embedded devices often use an interrupt multiplexer (or controller) peripheral that is normally included as an hardware block ("IP core") on the same chip. The disadvantage for a user is that at the point where the interrupt vector routine is called, all interrupt signals are still multiplexed together. The driver for a particular interrupt multiplexer will then query the underlying hardware multiplexer to identify which line was actually triggered and then forward the event to the handler registered for this interrupt.

Now, suppose the user wants to instruct *Avatar* to suppress a particular interrupt on the device (e.g., the timer), while still letting through the ones associated to important hardware events that need to be forwarded to the emulator. In this case, the proxy needs to take a decision based on the interrupt type which is unfortunately not available when the interrupt is received.

In this case, the user needs to disassemble the interrupt vector handler, and follow the code flow until the code of the interrupt controller driver branches into different functions that handle each device's interrupt. At this point, she can specify these program points to *Avatar* that can terminate the interrupt vector's execution and signal to the proxy that an interrupt has been identified. The proxy then sends the interrupt event to *Avatar*. Now the target backend of *Avatar* can suppress a particular interrupt by instructing the proxy to drop the corresponding event.

### 3.3.5 Replaying Hardware Interaction

It is quite common for a firmware to have several sections that require only a limited interaction with dedicated peripherals. In this case, the I/O operations can be recorded by *Avatar* and transparently replayed during the next execution of the firmware.

This allows the user to test the firmware without the bottleneck of the interaction with the physical device. In this mode of operation the firmware itself or parts of it (e.g., applications) can be significantly changed, as long as the order of I/O interactions is not modified. This is a major advantage over resuming a snapshot, which requires the full code path until the snapshot point to be executed to ensure that peripherals are in the state the snapshot expects them to be in.

## 3.4 Overcoming the limits of Full Separation

The techniques introduced in the previous section are enough to perform dynamic analysis on small portions of a firmware code. However, sometimes the internals and behavior of the system are completely unknown. In those cases, it can be very useful to perform the analysis on larger portions of the binary, or, in the extreme case, on the entire firmware.

In this case, the performance of *Avatar* running in full separation mode poses a great limitation to the usability of our framework. To overcome this problem, in this section we present two techniques designed to overcome the limits of full separation by moving part of the code to the physical device and part of the memory to the emulator. This results in a considerable reduction in the number of messages forwarded by *Avatar* between the emulator and the target, and therefore a large improvement in the overall performance of the analysis system.

### 3.4.1 Memory Optimization

Forwarding all memory accesses from the emulator to the target over a limited-bandwidth channel like UART or JTAG incurs in a heavy performance penalty. For example, in our experiments an average of five instructions per second were executed using the GDB stub through a 38400 baud UART connection.

The reason why memory operations need to be forwarded in the first place is that different embedded systems typically have different mappings of addresses to memory regions. Some of these memory regions are used for code (in RAM, ROM or Flash memories), stack and heap, but one or several regions will be used to access registers of physical peripherals through Memory-Mapped I/O (MMIO). In this case, any I/O operation on those areas is equivalent to sending and receiving data from an external device. If these address ranges are known, the user can configure *Avatar* to keep every read-only memory (such as the code segment) on the emulator. Read-write memory regions can also be marked as local to the emulator, but modifications to them need to be tracked by *Avatar* to be able to transfer those changes to the target at a later context switch. In fact, when an emulator-to-target context switch happens, all modified local memory ("dirty memory") needs to be copied to the target before the execution can resume on the embedded device.

However, in most of the cases the user does not know a priori which area of memory is assigned to I/O. For this reason, *Avatar* includes an automated memory optimization plugin that monitors the execution in the emulator and automatically identifies the regions that do not require access to the hardware. This includes the stack (easily identified by the execution of stack-related operations) and the code segment (identified by the values of the program counter). For any other area, *Avatar* starts by forwarding the read and write operations to the

| *Access type* | Read | Write | Cumulative |
|---|---|---|---|
| Code | 61,632 | - | 61,632 |
| Stack & data | 646 | 1,795 | 64,073 |
| I/O | 3,614 | 2,097 | 69,784 |

**Table 3.1:** Number of memory accesses grouped by memory regions for the HDD bootloader.

target device. It then keeps track of the values that are returned and applies a simple heuristic: if the target always returns the value that was previously written by the firmware code (or if it always returns the same value and it is never written by the firmware) then it is probably not assigned to a memory mapped device.

Table 3.1 shows an example of how many memory accesses could be saved by keeping memory regions local to the emulator: transferring the code region to the emulator would save 61,632 memory accesses (88%). Moving the stack and data region in local memory as well would save 64,073 memory accesses (92%). Only the I/O accesses cannot be moved to the emulator's memory.

### 3.4.2 Selective Code Migration

So far, we assumed that the firmware is either running entirely inside the emulator, or entirely on the embedded device. The user can instruct *Avatar* to switch from one mode to the other when certain conditions are met, but such context switches are time consuming.

In this section we present a fine-grained solution that allows the user to migrate only parts of the firmware code back to the target. This technique allows to overcome two limitations of the full-separation mode. Some code blocks need to be executed atomically, for example when there are timing constraints on the code. We will describe such a case in Section 3.6.1, where we encountered a function that read the timer twice and waited for the difference to be below a certain limit. Another example is when delays introduced by *Avatar* would lead the target in an invalid state. We encountered such a case during the DRAM initialization of the HDD, as shown in Section 3.6.1).

The second limitation addressed by selective code migration is related to the analysis performance. In fact, certain functions (e.g., polling loops and interrupt handlers) can be executed significantly faster when run natively on the target.

In the current *Avatar* prototype, code migration is supported at a function level. In this case, the code can be copied to its location in the target's memory without modification. Its exit points are then replaced by breakpoints, and the virtual machine register state is transferred from the emulator to the target. The

execution is resumed on the target until one of the exit breakpoints is triggered, and at that point the state is transferred back to the emulator. This transition is much faster than a complete context switch, since *Avatar* only needs to transfer few bytes and not the entire content of the memory.

Even though this simple technique is enough to circumvent critical code regions in several real world scenarios, it neglects some difficulties that may affect code migration. First, the code may read or write arbitrary memory locations associated, for example, with global variables. *Avatar* keeps track of those locations, copy their content over to the target before the execution, and copy written locations back after the execution. Second, the code may use instructions that change the control flow in unforeseen ways, like software interrupts, processor mode changes, and indirect jumps.

Our framework prototype addresses these issues by performing an on-the-fly static analysis. When a function is selected for code migration, *Avatar* disassembles its code using the `llvm-mc` disassembler. The result is then analyzed to identify critical instructions. In this way, we can predict memory accesses outside the function stack, compute the control flow of the code and verify that no instructions can escape from this computed control flow. As we describe in Section 3.6, this technique is sufficient to migrate small, atomic functions. However, we plan to extend the capabilities of the code migration system to apply transformations to the code. On the one hand, those transformations will allow to ensure that instructions which are not statically verifiable (e.g., indirect jumps) will not escape the proxy's sandbox. On the other hand, it can be used to track memory accesses, so that only the modified ("dirty") part of the state needs to be copied back from the target to the emulator when a context switch happens. Those critical instructions will be replaced with instrumentation code that calls functions in proxy, which will handle them in a safe way.

## 3.5 Extending Avatar

*Avatar*'s architecture is designed to be modular and its base framework can be easily customized to fit different analysis scenarios. We chose $S^2E$ as default *Avatar* emulator back-end because it offers many hooks and manipulation facilities on top of QEMU which facilitates the development of custom dynamic analysis plugins.

In this section, we show an example of an *Avatar* extension: we built upon its core capabilities to support selective symbolic execution. For this we add several features and plugins to the ARM port of $S^2E$. Moreover, we believe the symbolic execution engine provides a super-set of the capabilities needed to implement taint analysis, even though a targeted plugin could be needed to perform concrete data tracking and taint analysis in a more lightweight way.

In the rest of this section we describe the technique *Avatar* employs to fully exploit the symbolic engine of S$^2$E and perform selective symbolic execution on unmodified portions of firmware blobs. Moreover, we show how we use our extended version of S$^2$E in *Avatar* to dynamically detect potential control flow corruption vulnerabilities by injecting and tracking symbolic inputs.

### 3.5.1 Injecting Symbolic Values Into the Firmware's Execution Flow

In the field of program testing, symbolic execution is a technique employed to improve code coverage by using symbols as input data (instead of concrete values) and keeping track of constraints upon their manipulation or comparison (c.f. [SAB10]). The result of symbolic evaluation is an execution tree, where each path is a possible execution state that can be reached by satisfying the constraints associated to each symbolic value.

S$^2$E further develops this concept by performing selective symbolic execution, i.e., by restricting the area of symbolic execution to specific code portions and treating only specific input data as symbolic [CKC12]. This greatly helps to speedup the analysis process (as symbolic execution of code results in significant slowdowns) and to drive the exhaustive symbolic exploration into selected regions of code. This process requires *Avatar* to control the introduction of symbolic values into S$^2$E, in place of existing real values.

The remote memory interface between S$^2$E and *Avatar*, as introduced in Section 3.3, ensures that only concrete values reach the real hardware through *Avatar*. Symbolic values remain therefore confined to the emulation domain. If a symbolic value is about to be written to the target hardware, the remote memory interface in S$^2$E performs a forced concretization before forwarding it. Such symbolic value concretizations happen in two stages. First, all the constraints associated with the value are retrieved and evaluated by the integrated SAT-solver. Second, a single example value which satisfies all the constraints is forwarded to *Avatar* to be written on the target.

On the one hand, making *Avatar* handle only concrete values leaves it as a controller with a simpler external view of S$^2$E and avoids having to keep track of execution paths and paths conditions twice. On the other hand, this choice brings the minor drawback that *Avatar* has no direct control on symbolic execution, which is instead under the control of S$^2$E/KLEE.

We designed a simple plugin for detecting arbitrary execution conditions. It relies on the following heuristics as signs of possibly exploitable conditions:

- a symbolic address being used as the target of a load or store instruction,
- a symbolic address being leaked into the program counter (e.g., as the target of a branch),

- a symbolic address being moved into the stack pointer register.

In order to selectively mark some input data as symbolic, two different approaches can be taken: either modify the binary code (or the source code, if available) to inject custom instructions into the firmware, or dynamically instrument the emulation environment to specify the scope of symbolic analysis at run-time. The first approach requires some high-level knowledge of the firmware under analysis (e.g., access to source code) and the guarantee that injecting custom instructions into firmware code would not affect its behavior. Examples include the Android Dalvik VM, whose source code can be modified and rebuilt to enable transparent analysis of pristine Java bytecode with $S^2E$ [Kir].

Since we did not want to limit *Avatar* to this scenario, we decided to follow the second approach, which requires to extend the symbolic engine and the *Avatar* framework. Such extensions should know when symbolic execution has to be triggered and where symbolic values should be injected.

This choice leads to two major advantages:

- *Firmware Integrity*
  The binary code is emulated as-is, without injecting custom opcodes or performing recompilation. This guarantees that the emulated code adheres to the original firmware behavior (i.e., no side-effects or bugs are introduced by the intermediate toolchain)

- *Programmatic Annotation*
  The control and data flow of firmware emulation can be manipulated and annotated with symbolic meta-data in an imperative way. A high-level language (Lua) is used to dynamically script and interact with current emulation environment, as well as introducing and tracing symbolic meta-data.

For this we first completed the port of $S^2E$ to the ARM architecture in order to have complete symbolic execution capabilities, then we ported the Annotation plugin to the ARM architecture. The Annotation plugin lets the user specify a trigger event (e.g., a call/return to a specific subroutine or the execution of code at a specific address), and a Lua function to be executed upon the event. A simple API is then provided to allow for manipulation of the $S^2E$ emulation environment directly from the Lua code. *Avatar* provides direct channels to dynamically control the emulation flow via QMP command messages. These channels can also be used to inject Lua code at run-time, in order to dynamically generate annotations which depend on the current emulation flow and inject them back into $S^2E$. Once symbolic values are introduced in the execution flow, $S^2E$ tracks them and propagates the constraints.

Symbolic analysis via Lua annotations is intended to be used as a tool for late stage analysis, typically to ease the discovery of flaws in logic-handling code, with

hand-made Lua analysis code directly provided by the user. It can be employed in both full separation mode and context switching, as soon as code execution can be safely moved to the emulator (e.g., outside of raw I/O setup routines, sensors polling). This normally happens after an initial analysis has been done with *Avatar* to detect interesting code and memory mappings.

A similar non-intrusive approach has already been used in a x86-specific context, to test and reverse-engineer the Windows driver of a network card [CC10b]. To the best of our knowledge, however, this technique has never been applied before to embedded devices. In the context of firmware security testing, annotations can be used in a broad range of scenarios. In Section 3.6, we present how we applied this technique to different technologies and devices, to perform dynamic analysis of widespread embedded systems such as hard drives, GSM phones, and wireless sensors.

### 3.5.2 Symbolically Detecting Arbitrary Execution Conditions

When dealing with modern operating systems, an incorrect behavior in a user-space program is often detected because an invalid operation is performed by the program itself. Such operations can be, for example, an unauthorized access to a memory page, or the access to a page that is not mapped in memory. In those cases, the kernel would catch the wrong behavior and terminate the program, optionally triggering some analysis tools to register the event and collect further information that can later be used to identify and debug the problem. Moreover, thanks to the wide range of exploit mitigation techniques in place today (DEP, canaries, sandboxing and more), the system is often able to detect the most common invalid operations performed by userspace processes.

When dealing with embedded systems, however, detecting misbehavior in firmware code can be more difficult. The observable symptoms are not always directly pinpointed to some specific portion of code. For example, many firmware are designed for devices without a Memory Management Unit (MMU) or Memory Protection Unit (MPU) or are just not using them. In such a context, incorrect memory accesses often result in subtle data corruption which sometimes leads to erratic behaviors and rare software faults, such as random events triggering, UI glitches, system lock or slowdown [Cri82]. For this reason, it is common for embedded devices to have a hardware watchdog in charge of resetting the device execution in case of any erratic behavior, e.g., a missed reply to timed watchdog probes.

For these reasons, detecting incorrect execution inside the emulation is easier when some OS support can be used for co-operation (e.g., a *Blue Screen Of Death* interceptor for Windows kernel bugs is implemented in $S^2E$). On the other hand, catching such conditions during the emulation of an embedded device

firmware is bound to many system-specific constraints, and require additional knowledge about the internal details of the firmware under analysis.

However, *Avatar* does not rely on the knowledge of any specific operating system or the fact that a MMU is used. Instead, it aims at detecting a larger range of potentially critical situations which may result in control flow hijacking of firmware code, by using a technique similar to the one employed by AEG [ACHB11].

All three conditions may lead to false positives, when the variable is symbolic but strongly constrained. Therefore, once such a condition is detected the constraints imposed on the symbolic variables must be analyzed: the less constrained is the result, the higher is the chance of control flow corruption. Intuitively, if the constraints are very loose (e.g., a symbolic program counter without an upper bound) then the attacker may obtain enough control on the code to easily exploit the behavior. In contrast, tightly constrained symbolic addresses, such as a properly constrained pointer into a jump table, are not relevant for the purpose of security analysis.

When an interesting execution path is detected by the above heuristic, the state associated to the faulty operation is recorded and the emulation is terminated. At this point a test-case with an example input to reach this state is generated, and the constraints associated with each symbolic value are stored to be checked for false positives (i.e., values too strictly bound).

Automatically telling normal constraints apart from those that are a sign of a vulnerability is a complex task. In fact it would require knowledge of the program semantics that were lost during compilation (e.g., array boundaries). Such knowledge could be extracted from the source code if it is available, or might be extrapolated from binary artifacts in the executable itself or the build environment. In such cases, specific constraints could be fed into *Avatar* by writing appropriate plugins to parse them, for example by scanning debug symbols in a non-stripped firmware (e.g., a DWARF parser for ELF firmwares) or by reading other similar symbols information.

Finally, *Avatar* could highly benefit from a tighter coupling with a dynamic data excavator, helping to reverse engineer firmware data structures [CSXK08]. In particular, the heuristic proposed in Howard [SSB11] for recovering data structures by observing access patterns under several execution cycles could be easily imported into the *Avatar* framework. Both tools perform binary instrumentation on top of QEMU dynamic translation and make use of a symbolic engine to expand the analyzed code coverage area.

### 3.5.3 Limitations of state synchronization

Our current implementation of the synchronization between device state and emulator state works well in general, but is difficult in some special cases.

First it is difficult to handle DMA memory accesses in our current model. For example, the firmware can send a memory address to a peripheral and request data to be written there. The peripheral will then notify the firmware of the request's completion using an interrupt. Since *Avatar* does not know about this protocol between firmware and peripheral, it will not know which memory regions have been changed. On newer ARM architectures with caches, *data synchronization barrier* or *cache invalidation* instructions might be taken as hint that some memory region has been changed by DMA.

Second, if code is executed on the device, *Avatar* is currently incapable of detecting which regions have been modified. In consequence, whenever memory accesses of the code run on the device are not predictable by static analysis, we need to transfer the whole memory of the device back to the emulator on a device-to-emulator state switch. We plan to address this issue by using checksumming to detect memory region changes and minimize transferred data by identifying smallest changed regions through binary search.

Third, when *Avatar* performs symbolic execution, symbolic values are confined to the emulator. In case that a symbolic value needs to be concretized and sent to the device, a strategy is needed to keep track of the different states and I/O interactions that were required to put the device in that state. This can be performed reliably by restarting the device and replaying I/O accesses. While this solution ensures full consistency, it is rather slow.

## 3.6 Evaluation

In this section we present three case studies to demonstrate the capabilities of the *Avatar* framework on three different real world embedded systems. These three examples by no means cover all the possible scenarios in which *Avatar* can be applied. Our goal was to realize a flexible framework that a user can use to perform a wide range of dynamic analysis on known and unknown firmware images.

As many other security tools (such as a disassembler or an emulator), *Avatar* requires to be configured and tuned for each situation. In this section, we try to emphasize this process, in order to show all the steps a user would follow to successfully perform the analysis and reach her goal. In particular, we will discuss how different *Avatar* configurations and optimization techniques affected the performance of the analysis and the success of the emulation.

Not all the devices we tested were equipped with a debug interface, and the amount of available documentation varied considerably between them. In each case, human intervention was required to determine appropriate points where to hook execution and portions of code to be analyzed, incrementally building the knowledge-base on each firmware in an iterative way. A summary of the

| | Experiment 3.6.1 | Experiment 3.6.2 | Experiment 3.6.3 |
|---|---|---|---|
| Target device | Hard disk | ZigBee sensor | GSM phone |
| Manufacturer and model | *undisclosed* | Redwire Econotag | Motorola C118 |
| System-on-Chip | unknown | MC13224 | TI Calypso |
| CPU | ARM966 | ARM7TDMI | ARM7TDMI |
| Debug access | Serial port | JTAG | JTAG |
| Analyzed code | Bootloader | ZigBee stack | SMS decoding |
| Scope of analysis | Backdoor detection | Vulnerability discovery | Reverse engineering |

**Table 3.2:** Comparison of experiments described in Section 3.6.

main characteristics of each device and of the goal of our analysis is shown in Table 3.2.

### 3.6.1 Analysis of the Mask ROM Bootloader of a Hard Disk Drive

Our first case study is the analysis of a masked ROM bootloader and the first part of the secondary bootloader of a hard disk drive.

The hard disk we used in our experiment is a commercial-off-the-shelf SATA drive from a major hard disk manufacturer. It contains an ARM 966 processor (that implements the ARMv5 instruction set), an on-chip ROM memory which contains the masked ROM bootloader and some library functions, an external serial flash that is connected over the SPI bus to the processor, a dynamic memory (SDRAM) controller, a serial port accessible through the master/slave jumpers, and some other custom hardware that is necessary for the drive's operation. The drive is equipped with a JTAG connection, but unfortunately the debugging features were disabled in our device. The hard drive's memory layout is summarized in Figure 3.4.

The stage-0 bootloader executed from mask ROM is normally used to load the next bootloader stage from a SPI-attached flash memory. However, a debug mode is known to be reachable over the serial port, with a handful of commands available for flashing purposes. Our first goal was to inject the *Avatar* stub through this channel to take over the booting process, and later use our framework for deeper analysis of possible hidden features (e.g., backdoors reachable via the UART).

The first experiment we performed consisted of loading the *Avatar* stub on the drive controller and run the bootloader's firmware in full separation mode.

**Figure 3.3:** The disk drive used for experiments. The disk is connected to a SATA (Data+Power) to USB interface (black box on the right) and its serial port is connected to a TTL-serial to USB converter (not shown) via the 3 wires that can be seen on the right.

This mimics what a user with no previous knowledge of the system would do in the beginning. In full separation mode, all memory accesses were forwarded through the *Avatar* binary protocol over the serial port connection to the stub and executed on the hard drive, while the code was interpreted by $S^2E$. Because of the limited capacity of the serial connection, and the very intensive I/O performed at the beginning of the loader (to read the next stage from the flash chip), only few instructions per second were emulated by the system. After 24 hours of execution without even reaching the first bootloader menu, we aborted the experiment.

In the second experiment we kept the same setting, but we used the memory optimization plugin to automatically detect the code and the stack memory regions and mark them as local to the emulator. This change was enough to reach the bootloader menu after approximately eight hours of emulation. Though considerably faster than in the first experiment, the overhead was still unacceptable for this kind of analysis.

Since the bottleneck of the process was the multiple read operations performed by the firmware to load the second stage, we configured *Avatar* to replay the hardware interaction from disk, without forwarding the request to the real hardware. In particular, we used the trace of the communication with the flash memory from the second experiment to extract the content of the flash memory, and dump it into a file. Once the read operations were performed locally in the emulator, the bootloader menu was reached in less than four minutes.

At this point, we reached an acceptable working configuration. In the next experiment, we show how *Avatar* can be used in conjunction with the symbolic execution of $S^2E$ to automatically analyze the communication protocol of the

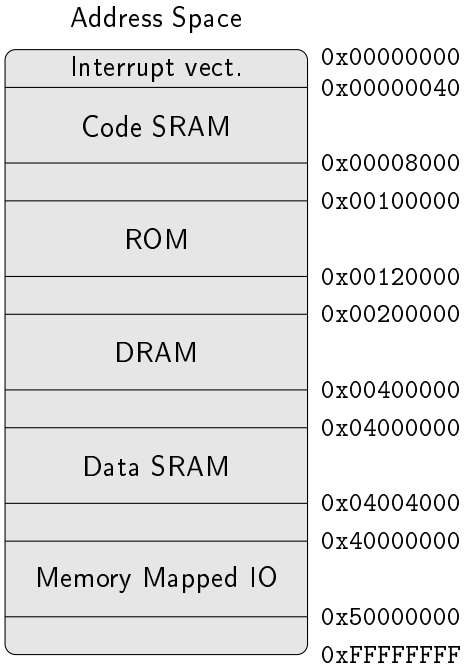


**Figure 3.4:** Hard drive memory layout.

hard drive's bootloader and detect any hidden backdoor in it.

We configured *Avatar* to execute the hard drive's bootloader until the menu was loaded, and then replace all data read from the serial port register by symbolic values. As a result, $S^2E$ started exploring all possible code paths related to the user input. This way, we were able to discover all possible input commands, either legitimate or hidden (which may be considered backdoors), that could be used to execute arbitrary code by using $S^2E$ to track when symbolic values were used as address and value of a memory write, and when the program counter would become symbolic. With similar methodologies, a user could use symbolic execution to automatically discover backdoors or undocumented commands in input parsers and communication protocols.

In order to conduct a larger verification of the firmware input handler, we were also able to recover all the accepted commands and verify their semantics. Since the menu offered a simple online `help` to list all the available commands, we could demonstrate that *Avatar* was indeed able to automatically detect each and all of them (the complete list is reported in Table 3.3). In this particular device, we verified that no hidden commands are interpreted by the firmware and that a subset of the commands can be used to make arbitrary memory modifications or execute code on the controller, as documented.

However, we found that the actual protocol (as extracted by symbolic analysis) is much looser than what is specified in the `help` menu. For example the argument

| | |
|---|---|
| **DS** | Use a minimal version of the Motorola S-Record binary data format to transmit data to the device |
| **AP <addr>** | Set the value of the address pointer from the parameter passed as hexadecimal number. The address pointer provides the address for the read, write and execute commands. |
| **WT <data>** | Write a byte value at the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state. |
| **RD** | Read a byte from the memory pointed to by the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state. |
| **GO** | Execute the code pointed to by the address pointer. The code is called as a function with no parameters, to execute Thumb code one needs to specify the code's address + 1. |
| **TE** | Switch the terminal echo state. The terminal echo state controls the verbosity of the read and write commands. |
| **BR <divisor>** | Set the serial port baud rate. The parameter is the value that will be written in the baud rate register, for example "A2" will set a baudrate of 38400. |
| **BT** | Resume execution with the firmware loaded from flash. |
| **WW** | Erase a word (4 bytes) at the address pointer and increment address pointer. |
| **?** | Print the help menu showing these commands. |

**Table 3.3:** Mask ROM bootloader commands of the hard drive. In the left column you can see the output of the help menu that is printed by the bootloader. In the right column a description obtained by reverse engineering with symbolic execution is given.

of the 'AP' command can be separated by any character from the command, not only spaces. It is also possible to enter arbitrarily long numbers as arguments, where only the last 8 digits are actually taken into account by the firmware code.

After the analysis of the first stage was completed, we tried to move to the emulation of the second stage bootloader. At one point, in what turned out to be the initialization of the DRAM, the execution got stuck: the proxy on the hard drive would not respond any more, and the whole device seemed to have crashed. Our guess was that the initialization writes the DRAM timings and needs to be performed atomically. Since we already knew the exact line of the crash from the execution trace, it was easy to locate the responsible code,

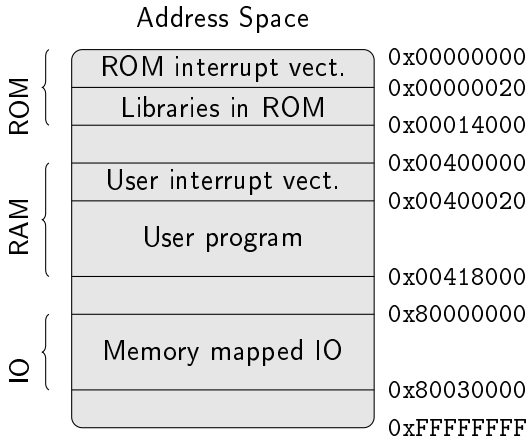


**Figure 3.5:** Econotag memory layout (respective scales not respected).

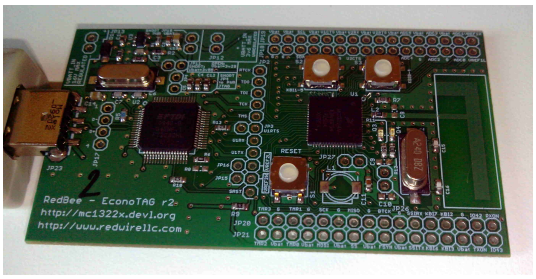

**Figure 3.6:** The Econotag device. From left to right: the USB connector, serial and JTAG to USB converter (FTDI), Freescale MC13224v controller and the PCB 2.4 GHz antenna.

isolate the corresponding function, and instruct *Avatar* to push its code back to be executed natively on the hard drive.

In a similar manner, we had to mark few other functions to be migrated to the real hardware. One example is the timer routine, which was reading the timer value twice and then checked that the difference was below a certain threshold (most probably to ensure that the timer read had not been subject to jitter). Using this technique, in few iterations we managed to arrive at the final *Avatar* configuration that allowed us to emulate the first and second stages up to the point in which the disk would start loading the actual operating system from the disk's platters.

### 3.6.2 Finding Vulnerabilities Into a Commercial Zigbee Device

The Econotag, shown in Figure 3.6, is an all-in-one device for experimenting with low power wireless protocols based on the IEEE 802.15.4 standard [IEE06],

such as Zigbee or 6lowpan [MKHC07]. It is built around the MC13224v System on a Chip from Freescale. The MC13224v [Red] is built upon an ARM7TDMI microcontroller, includes several memories, peripherals and has an integrated IEEE 802.15.4 compatible radio transceiver. As it can be seen in Figure 3.5, the device includes 96KB of RAM memory, 80 KB of ROM and a serial Flash for storing data. The ROM memory contains drivers for several peripherals as well as one to control the radio, known as $MACA$ (MAC Accelerator), which allows to use the dedicated hardware logic supporting radio communications (e.g., automated ACK and CRC computation).

The goal of this experiment is to detect vulnerabilities in the code that process incoming packets. For this purpose, we use two Econotag devices and a program from the Freescale demonstration kit that simulates a wireless serial connection (wireless UART [Fre11a]) using the $Simple\ MAC$ (SMAC [Fre11b]) proprietary MAC layer network stack. The program is essentially receiving characters from its UART and transmitting them as radio packets as well as forwarding the characters received on the radio side to its serial port. Two such devices communicating together essentially simulate a wireless serial connection.

The data received from the radio is buffered before being sent to the serial port. For demonstration purposes, we artificially modified this buffer management to insert a vulnerability: a simple stack-based buffer overflow. We then compiled this program for the Econotag and installed it on both devices.

*Avatar* was configured to let the firmware run natively until the communication between the two devices started. At this point, *Avatar* was instructed to perform a context switch to move the run-time state (registers and data memory) of one of the devices to the emulator. At this point, the execution proceeded in full separation mode inside the emulator using the code loaded in ROM memory (extracted from a previous dump), and the code loaded in RAM memory (taken from the application). Every I/O access was forwarded to the physical device through the JTAG connection.

The emulator was also configured to perform symbolic execution. For this purpose, we used $annotations$ to mark the buffer that contains the received packet data as symbolic. Then, we employed a state selection strategy to choose symbolic states which maximize the code coverage, leading to a thorough analysis of the function.

On the first instruction that uses symbolic values in the buffer, $S^2E$ would switch from concrete to symbolic execution mode. Execution will fork states when, for example, conditional branches that depend on such symbolic values are evaluated. After exploring 564 states, and within less than a minute of symbolic execution, our simple $arbitrary\ execution\ detection\ module$ detected that an unconstrained symbolic value was used as a return address. This confirmed the detection of the vulnerability and also provided an example of payload that triggers the vulnerability.

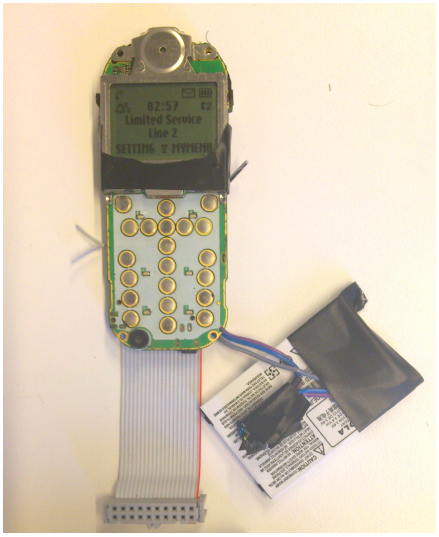


**Figure 3.7:** The Motorola C118. The clip-on battery (on the right) has been wired to the corresponding power pins, while the ribbon cable is connected to the JTAG pads reachable on the back (not shown).

We also used *Avatar* to exhaustively explore all possible states of this function on a program without the injected vulnerability, and confirmed the absence of control flow corruption vulnerabilities that could be triggered by a network packet (that our simple arbitrary execution detection module could detect).

### 3.6.3 Manipulating the GSM Network Stack of a Common Feature Phone

Our final test-case is centered on the analysis of the firmware of a common GSM feature phone. In contrast with most recent and advanced mobile phones and smartphones, feature phones are characterized by having one single embedded processor for both the network stack (i.e., GSM baseband capabilities) and the Human-to-Machine Interface (HMI: comprising the main Graphical User Interface, advanced phone services, and miscellaneous applications). As such, there is no clear code separation between different firmware sections. On these phones, typically a real-time kernel takes care of scheduling all the tasks for the processes currently in execution. These are executed in the same context and have shared access to the whole physical memory as well as memory-mapped I/O.

GSM baseband stacks have already been shown to have a large potentially exploitable attack surface [Wei12]. Those stacks are developed by few companies worldwide and have many legacy parts which were not written with security in mind, and in particular were not considering attacks coming from the GSM

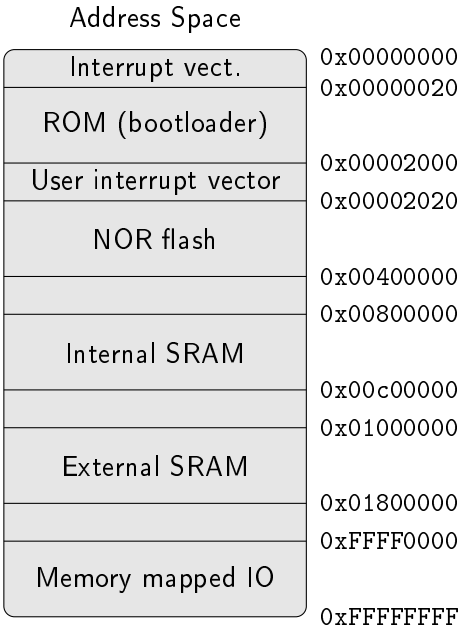


**Figure 3.8:** Motorola C118 memory layout (respective scales not respected).

infrastructure [Wel].

For our experiment, we used a Motorola C118, which is a re-branded version of the Compal E88 board also found in other Motorola feature phones. This board makes use of the *Texas Instruments "Calypso"* digital baseband, which is composed of a mask-ROM, a DSP for GSM signal decoding, and a single ARM7TDMI processor. It also includes several peripherals such as an RTC clock, a PWM generator for controlling the lights and buzzer as well as a memory mapped UART as shown in Figure 3.8. Some board models have JTAG and UART ports available, which are from time to time left enabled by manufacturers to simplify servicing devices. In our case, we gained access to the JTAG port and used an adapter to bridge communication between *Avatar* and the hardware, as shown in Figure 3.7.

Some specification documents on the *Calypso* chipset have been leaked in the past, leading to the creation of home-brew phone OS that could be run on such boards. As part of the Osmocom-BB project, most of the platform has been reversed and documented, and it is now possible to run a free open-source software GSM stack on it [osm]. However, we conducted our experiments on the original Motorola firmware, in order to assess the baseband code of an unmodified phone. Moreover, as the GSM network code is provided as a library by the baseband manufacturer, there is an higher chance that flaws affecting the library code would also be present in a broader range of phones using baseband chips from that same vendor.

The phone has a first-stage bootloader executed on hardware reset, which can be used to re-flash the firmware. After phone setup, execution continues to the main firmware, which is mainly composed of the Nucleus RTOS, the TI network stack library, and of third-party code to manage the user interface. The phone bootloader can be analyzed using *Avatar* in a similar way as the one already described for the hard disk in Section 3.6.1 to discover flashing commands, hidden menus and possible backdoors. However, the bootloader revealed itself to be simpler than the hard drive one, supporting only a UART command to trigger firmware flashing and executing the flashed firmware, or continuing execution after a timeout expiration.

For this reason, we focused on the analysis of the GSM network stack, and in particular on the routines dedicated to SMS decoding. It has already been shown in the past how maliciously crafted SMS can cause misbehavior, ranging from UI issues to phone crashes [MGS11]. However, due to the lack of a dynamic analysis platform to analyze embedded devices, previous studies relied on blind SMS fuzzing. Our experiment aims at improving the effectiveness of SMS fuzzing to detect remotely exploitable execution paths.

In this scenario, *Avatar* was configured to start the execution of the firmware on the real device, and switch to the emulator once the code reached the SMS receiving state (e.g., by sending a legitimate SMS to it through the GSM network). *Avatar* was then used to selectively emulate and symbolically explore the decoding routines. As a result of this exploration, a user is able to detect faulty conditions, to determine code coverage due to different inputs and to recover precise input constraints to drive the firmware execution into interesting areas.

In this context, *Avatar* uses the JTAG connection to stop the execution on the target and later perform all synchronization steps between the emulator and the target. All memory and I/O accesses through JTAG are traced by *Avatar* to let the user identify address mappings. When the phone reaches the SMS receiving state, a target-to-emulator context switch happens and the phone's state is transferred into $S^2E$. Using address mapping information previously recovered through *Avatar*, just the relevant memory is moved into $S^2E$ (e.g., portions of code and the execution stack), while remaining memory is kept on the target and forwarded on-the-fly by *Avatar* (e.g., I/O regions). On this device, no selective code migration was required.

Using this *Avatar* configuration, the SMS payload can be intercepted in memory and marked as symbolic by employing the techniques shown in Section 3.5. In particular, we wrote Annotation functions to be triggered before entering the decoding routines and we then proceeded to selectively mark some bytes of their input arguments as symbolic. The $S^2E$ plug-in for Arbitrary Execution Detection has been employed to isolate interesting vulnerable cases, while other execution paths were killed upon reaching the end of the decoding function.

The symbolic execution experiments have been performed over several days, with

the ones with larger number of symbolic inputs taking up to 10 hours before filling up 60 GB of available memory. In such case, we observed more than 120,000 states being spawned according to different constraints solving. Unfortunately, and contrary to the other experiments, the GSM network stack proved to be way too complex to be symbolically analyzed without prior knowledge on the high-level structure of the code. The analysis was clobbered by an explosion of possible states due to many forks happening in pointer-manipulating loops. *Avatar* was able to symbolically explore 42 subroutines executed during SMS decoding, without detecting any exploitable conditions. However, it was able to highlight several situations of user-controlled memory load, which were unfortunately too strictly constrained to be exploited, as discussed in Section 3.5.2.

State explosion is a well-known limitation of symbolic execution. To mitigate the problem, a user may need to define heuristics to avoid an excessive resource consumption. This could be done, for example, by employing more aggressive state selectors to enhance code coverage, and actively prune states by looking at loops invariants [SW07]. However, this optimizations are outside the scope of our work. The objective of our experiments are, in fact, limited to prove that *Avatar* can be used to perform dynamic analysis of complex firmware of embedded devices.

# Chapter 4

# Implementation and Implications of a Stealth Hard-Drive Backdoor

*This chapter is based on a publication which has been presented at the 29th Annual Computer Security Applications Conference (ACSAC) in 2013 [ZKB+13].*

In the previous chapter, we presented *Avatar*, a dynamic analysis framework for firmware. We have shown its usefulness on three test cases. Here, we want to exand on one of the test cases, the hard drive, and perform an analysis of the whole firmware, not just the boot ROM. As we will explain later, there are some intricacies in injecting *Avatar*'s GDB stub into the firmware and preventing it from being overwritten. However, in the end, the tracing features of the framework proved very helpful to identify possible locations where data blocks can be intercepted in the firmware.

We also want to highlight the topicality of firmware backdoors. In January 2014, Just about a month after this work was presented, documents leaked by whistleblower Edward Snowden revealed that the National Security Agency (NSA) had been working on a firmware implant for hard drives, codenamed "IRATEMONK" [Sch14, AGG+15]. From our understanding, this backdoor is used to inject a rootkit into the Master Boot Record (MBR). Code in the MBR runs at the very beginning of the boot process in the highest privilege level. Even if the disk is completely erased, the rootkit can be re-injected into the MBR by the firmware and install itself from there. As we will show, the backdoor presented in this chapter works differently, and does not execute any code on the main CPU.

In our original paper we claim that "the difficulty of implementing such an attack is not limited to the area of government cyber-warfare; rather, it is well within the

reach of moderately funded criminals, botnet herders and academic researchers". This claim is bolstered today by the work of another private security researcher. Jeroen Domburg reverse-engineered at the same time, independently from us, a hard drive firmware and presented his work at a hacking conference in July 2013 [aS], just when this work was accepted to be presented at ACSAC 2013. Finally, Kaspersky reported in February 2015 to have discovered a malware that is capable of infecting hard drive firmware [kas15].

## 4.1 Introduction

Rootkits and backdoors are popular examples of malicious code that allow attackers to maintain control over compromised machines. They are used by simple botnets as well as by sophisticated targeted attacks, and they are often part of cyber-espionage tools designed to remain undetected and collect information for a long period of time.

Traditionally, malicious code targets system utilities, popular network services or components of the operating system. However, in a continuous effort to become more persistent and avoid detection, the target of the infection has shifted from software components towards more low-level elements, such as bootloaders, virtual-machine hypervisors, computer BIOS, and recently even the hardware itself.

The typical hardware-based threat scenario involves a malevolent employee in the manufacturing process or a compromised supply chain. In addition, many devices from trusted parties have been known to contain rootkits for copyright protection [HF06] or lawful interception capabilities in network devices [BFS04, Cro10]. Recent reports of hard drives shipping with viruses [Max13] show that such threats are also realistic in the context of storage devices. In this chapter, we will demonstrate that it is not even necessary to have access to the manufacturer or to the supply chain in order to compromise a hard drive's firmware. Instead, a firmware backdoor can be installed by, e.g., traditional malware after the operating system has been compromised.

From the attacker's point of view, a drawback of hardware backdoors is the fact that they are highly hardware dependent, requiring customization for each targeted device. This has made hardware backdoors less generic and less attractive than more traditional operating-system backdoors. However, the hard-drive market has now shrunk to only three major manufacturers, with Seagate and Western Digital accounting for almost 90% of all drives manufactured [bac12]. While drive firmwares may vary across product lines, porting a backdoor from one model to another of the same manufacturer should require only a limited amount of work, making backdoors on hard drives an attractive attack vector.

So far, malicious hardware has typically been used as a stepping stone to compromise other system components: for example, by exploiting the auto-run functionality, filesystem vulnerabilities [Lar11], or DMA capabilities on systems lacking properly configured I/O Memory Management Units (IOMMU). In such cases, malicious code on the operating system is simply *bootstrapped* from the hardware device. Then, to perform its operation, the malware propagates and infects the OS kernel, using the compromised hardware only as a way to survive reinstallation and software updates. However, as soon as malicious code "leaves" the firmware and moves to the system memory, it breaks cover. Therefore, such malware can be detected and prevented by kernel- or hardware-supported integrity mechanisms, such as Copilot [PJFMA04].

In this chapter, we describe how an attacker can overcome the above limitations by leveraging a storage firmware backdoor. Such a firmware backdoor does not require any modification to the operating system. The backdoor is, therefore, less intrusive and less dependent on other layers (e.g., OS, applications, and filesystem). As a consequence, it cannot be detected by existing mechanisms that guarantee OS integrity [PJFMA04, HDK+11].

As a proof of concept, we present a Data Exfiltration Backdoor ("DEB") that allows an attacker to remotely retrieve and modify any data stored in the device. A DEB allows a bi-directional communication channel to be established between the attacker and the storage device that potentially resides in a data center well outside the attacker's reach. As most Internet-based services, such as web forums, blogs, cloud services or Internet banking, eventually need to read and write data to disk, a DEB can be used to remotely exfiltrate data from such services. The rationale of this *data-replacement backdoor* is that the attacker can piggy-back its communications with the infected storage device on disk reads and writes. Indeed, the attacker can issue a specific command by encapsulating it in normal data which is to be written to a block on a compromised hard drive. This command makes the malicious firmware replace the data to be written with the data of an *arbitrary* block specified by the attacker. In a second step, the attacker can then request the block that was just written and therewith, effectively, retrieve the content of any block on the hard-drive. We also discuss a number of challenges that arise with this technique, and show how the attacker can overcome them (e.g., data alignment and cache issues).

**Threat Model** In our threat model, an attacker has compromised an off-the-shelf computer. This machine may have been initially infected with a malware by a common attack such as a drive-by-download or a malicious email attachment. Then the malware infects the machine's hard drive firmware by abusing its firmware update mechanisms. Finally, the OS part of the malware removes itself from the machine, and future malicious behavior becomes completely "invisible" to the OS, anti-virus or forensics tools. Following such an infection, the

malware can keep control of the machine without being detected even if the drive is formatted and the system re-installed.

We show in this chapter that, surprisingly, the above attack requires the same amount of effort and expertise as the development of many existing forms of professional malware (e.g., large scale botnets). Moreover, we claim that this attack is well within the capabilities of current cyber-espionage tools. Finally, we note that this threat model applies to dedicated hosting providers, since an attacker could temporarily lease a dedicated server and infect an attached hard drive via a malicious firmware update. A subsequent customer leasing a server with this infected drive would then be a victim of this attack.

## 4.2 Backdooring a Commercial Off-The-Shelf Hard Drive

In this section we describe how we inserted a backdoor into the firmware of a stock hard drive.

### 4.2.1 Modern Hard-Drive Architecture

The software and system architecture described here are specific to the drive we analyzed. However, we observed that it is almost identical for two distinct drives from one product family of the same manufacturer, and a brief look at one drive from another major manufacturer revealed a very similar architecture.

**Physical Device** A hard disk is a set of rigid magnetic disks aligned on a spindle, which is rotated by a motor. A rotary actuator structure moves a stack of heads relative to concentric tracks on the surface of the disks. The entire apparatus is contained in a tightly sealed case. A micro-controller takes care of steering the motors and translating the higher-level protocol that a computer uses to communicate with the disk to and from a bitstream, which is processed by specialized hardware (a DSP or FPGA) and fed to the heads [CFRN95]. Today, hard disks interface with other systems mostly through Serial ATA (SATA) and Small Computer Systems Interface (SCSI) buses, although bridge chips might translate to other buses, such as USB. Parts of those protocols are typically handled directly in hardware.

**Execution Environment** Like many embedded systems, this hard drive is based on a custom System on Chip (SoC) design. This SoC is built around an ARM966 CPU core, a read-only memory (ROM) containing a "mask ROM" bootloader, internal SRAM memories, an external serial FLASH (accessed via an

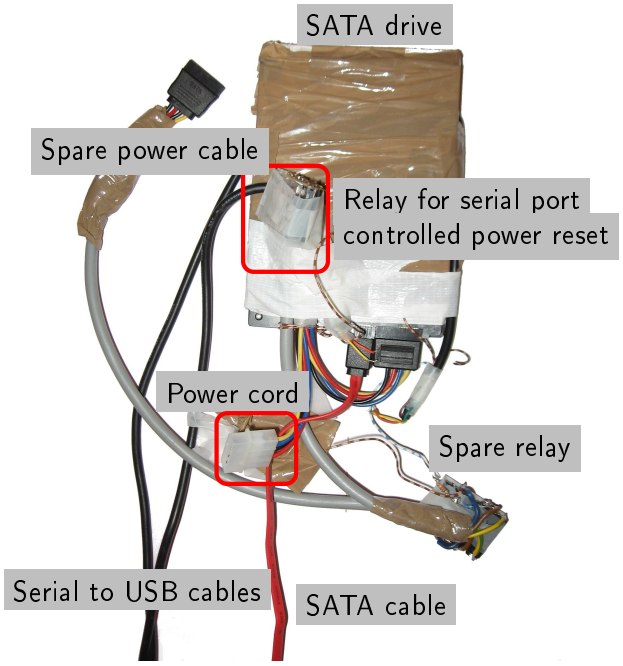


**Figure 4.1:** Custom backdoor development kit. This apparatus was built to reset the drive, allowing easy scripting and automated tasks. One USB to serial cable controls the relay, the second is connected to the serial port of the drive. The SATA cable is connected through a USB-SATA adapter for backdoor development. It is then directly connected to a computer motherboard for the field tests.

SPI bus), and an external DRAM memory. This DRAM is the largest memory and is used to cache data blocks read from or written to disk as well as a part of the firmware code that does not fit into the SRAM.

Interestingly, this hard drive also provides a serial console accessible through a physical serial port on the drive's Master/Slave jumper pins.

**Software Architecture and Boot Sequence** The bootloader in mask ROM is executed immediately after the CPU resets and loads a reduced boot firmware from the serial FLASH chip. The boot firmware has the capability to initialize the hardware to a sufficient degree to load the main firmware from the magnetic disks. However, it does not implement the full SATA protocol that this hard drive uses to talk to the computer.

Finally, the main firmware is loaded into memory from a reserved area of the disk (not user accessible) and then executed. Additional *overlays*, providing non-default functionality, can be loaded on demand from the reserved area. For

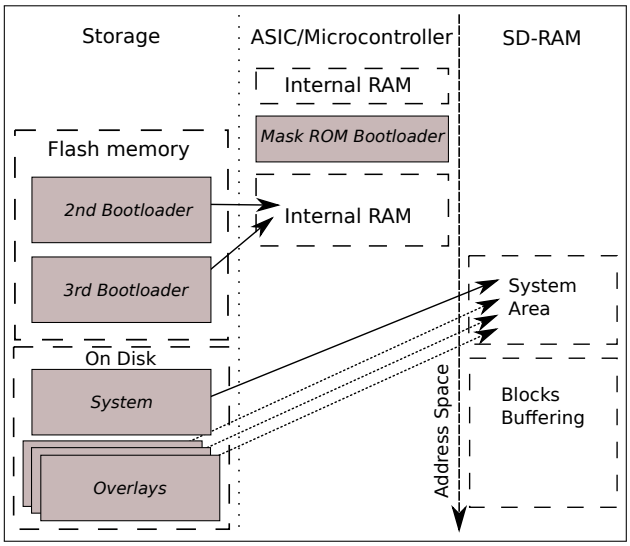


**Figure 4.2:** Overview of a hard drive's architecture.

example, a diagnostic menu available through the serial console is in overlays "4" and "5". The memory layout at run-time is depicted in Figure 4.2.

As our hard drive has a SATA bus, read and write requests to it are encapsulated in the ATA protocol. This is a simple master-slave protocol where the computer will always send a request, to which the hard drive replies with a response.

Inside the hard drive's firmware, five components take care of processing data: the interrupt handlers process hardware events, the SATA task processes data from the SATA port, the cache task manages the cache memory and evicts blocks from the cache, the read-write task transfers data to and from the disk platters, and the management task handles diagnostic menu commands and background activities.

**Analysis Techniques** Knowledge about the system was acquired from publicly available information (e.g., [hdd13]) and by reverse-engineering a hard drive in our lab. While the firmware (except for the mask ROM bootloader) is contained in update files, their format is not obvious and the header format was not documented. Fortunately, the diagnostic menu allows parts of memory to be dumped while the system is loaded. Thus, it proved easier to dump the firmware of the running hard drive through this menu than recovering it from the firmware update binary.

The mask ROM bootloader contains another menu, which can be accessed at boot time on the serial console. This menu provides a means to read and write the memory contents before the boot firmware is loaded. We therefore designed a

small GNU Debugger (GDB) stub that we injected into the hard drive's memory.

Inconveniently, our target hard drive's ARM 966 [ARM04] core lacks hardware debug support. Therefore, we relied purely on software breakpoints, rather than on hardware breakpoints or single-stepping. In this context, software breakpoints are essentially instructions that trigger a data abort interruption. By hooking into that interrupt vector's handler and replacing instruction by a breakpoint, one can implement a debugger stub fully in software.

If a software breakpoint is overwritten prior to it being reached, e.g., because the firmware loads new code, the breakpoint will never be triggered. In addition, we have observed that interrupt vectors or the debugger code itself can be overwritten by the firmware. To work around these problems, because of the lack of hardware breakpoints and watch-points, we manually identified all sections of code that load new code and hooked these functions to keep our debugger from disconnecting.

Finally, because setting a software breakpoint requires to modify instructions, it was not possible to put breakpoints on the ROM memory that contains the first bootloader and many other important library functions.

Our debugger stub itself requires only $3.4$ kB of memory, and it can be easily relocated to a new address. It communicates with a GDB instance over the serial port while still allowing the firmware's debug messages to be printed on the serial port. As the stub is stateless, it does not require any permanent storage of information. Complex debugging features, such as the bookkeeping required for breakpoints, are managed on the reverse engineer's workstation by GDB.

### 4.2.2 Developing Malicious Payloads

Our main goal in designing a proof-of-concept compromised hard-drive firmware is to be able to modify blocks as they are read from or written to the disk. Hooking into $write$ requests allows the backdoor to read and tamper with data blocks in the write buffer before they are written to the disk. In particular, we use a sequence of bytes in the first few bytes of a block, as a $magic\ value$. When this $magic\ value$ is detected by the backdoored firmware, predefined actions of the backdoor will be triggered.

**Hooking Writes in the Firmware** A write operation in a modern hard drive specifies the logical block number to write to (LBA), the number of blocks to write, and the data to be written. This information is encoded in ATA commands and transmitted to the hard drive through the Serial ATA connection.

On the hard drive we reverse engineered, specialized hardware is responsible for receiving the ATA messages and notifying the firmware by raising an interrupt. The firmware then performs the action corresponding to the $opcode$ field of the

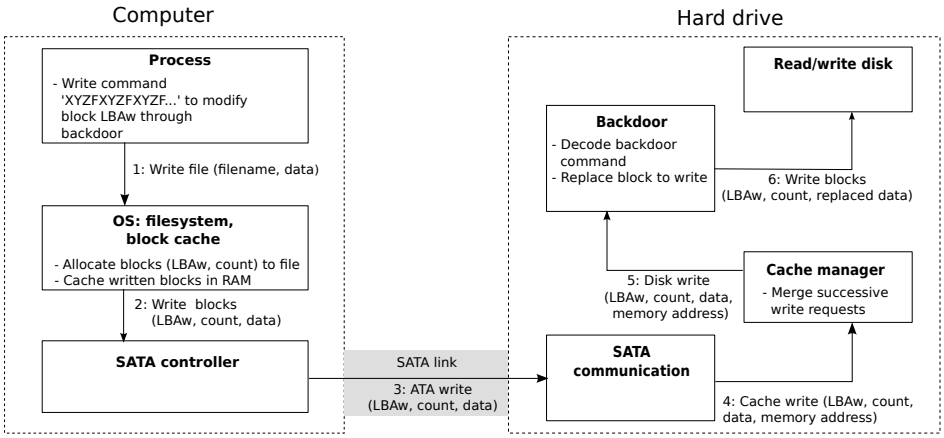


**Figure 4.3:** Call sequence of a write operation on the hard drive.

ATA message. In a *write DMA extended* ATA command, the data is then passed to the cache management task. This task keeps the received data blocks in volatile low-latency memory. When contiguous blocks are received, the firmware aggregates these blocks in memory. Eventually, the blocks will be evicted from cache memory, either because the cache is filled with newer data, or because a task commits them to the hard drive. Finally, the blocks will be passed to the read/write task, which takes care of positioning the head on the right track of the platter, and writes the data to the magnetic storage.

Figure 4.3 shows the sequence of the operations inside the hard drive. Our backdoor inserts itself in the call chain between the cache manager and the read/write task. By hooking writes after the cache, we ensure that the performance overhead remains low. At this point ATA commands have already been acknowledged, thus, the overhead of searching for the magic command in a block is less apparent to the user.

**Reading Blocks from inside the Firmware** Reading blocks inside the firmware proved to be harder than modifying writes. In order to read an arbitrary block, the modified firmware has to invoke a function providing several structured parameters. In our prototype implementation, this operation seems to trigger some internal side effect that makes the firmware unstable when multiple consecutive read operations are performed by our code.

**Update Packaging and Final Payload** Thanks to the debugger and the full firmware image, we were able to understand the firmware update format. We then generated a modified firmware update file that includes the original firmware infected with our proof-of-concept malicious code. Such a firmware update file

can then be programmed to the disk with the manufacturer's firmware update tool, which could be done by a malware with administrator rights. The backdoor will then be permanently installed on the drive.

With the current state of our reverse engineering of the hard drive, we can reliably hook write commands received by the hard drive and modify the data to be written to the magnetic platter. The backdoor can also read and exfiltrate arbitrary blocks, but it is not stable enough to retrieve multiple blocks from the disk. A more stable implementation would allow the full port of the Data Exfiltration Backdoor that we will present in Section 4.3. We could invest more time to try to solve the bug in our code, but there are few incentives to do so as our aim is to demonstrate the feasibility of such attacks rather than to develop a weaponized exploit for the hard drive.

However, the current state is sufficient to fully implement more straightforward attacks. For example, we can re-implement the famous backdoor presented by Ken Thompson in *Reflections on Trusting Trust* [Tho84]. In this lecture Thompson presented a compiler that inserts a backdoor while compiling the UNIX `login` command, allowing the password check to be bypassed. Similarly a compiler would transmit such a functionality when compiling a compiler. A malicious drive version of the `login` program backdoor simply detects a write to the disk of a critical part of the `login` binary and replace the code by a malicious version of the `login` binary.

### 4.2.3 Evaluation of the backdoor

We performed an *overhead test* to measure the impact of the backdoor under worst-case hard-drive operation. Indeed, if the backdoored firmware introduced significant overhead, this may alert a user of an anomaly.

This experiment is performed on the hard drive with the firmware backdoor described in Section 4.2.2, on an Intel Pentium E5200 2.5 GHz desktop computer equipped with 8 GB of physical memory. The hard drive was connected over internal SATA controller (Intel 82801JD/DO (ICH10 Family) 4-port SATA/IDE Controller).

**Overhead Test** We measured the write throughput on the test machine using IOZone [IOZ13]. As the backdoor functionality is only activated during writes, we use the IOZone *write-rewrite* test. We compare the write throughput obtained on the system running the unmodified hard drive firmware with the one running the backdoored firmware.

We perform the test with the IOZone `o_direct` option set to compare the results when the filesystem cache is not present. Most applications make use of the filesystem buffer cache to optimize access to the hard drive. However, with

the cache enabled, our experiments showed it was impossible to distinguish the performance of the modified firmware from the original one. Hence, we emulate, as best as we can, a suspicious user attempting to detect hard-drive anomalies by testing the direct throughput.

**Table 4.1:** Filesystem-level write-throughput.

| | Write test | |
|---|---|---|
| | Mean (MB/s) | 95% CI |
| With backdoor | 37.57 | [37.56; 37.59] |
| Without backdoor | 37.91 | [37.89; 37.94] |

We perform 30 iterations of the experiment, with a 30 second pause between successive iterations. For each set of values measured, we compute 95%-confidence intervals using the t-distribution. Table 4.1 shows the comparison of the write throughputs of the hard drive with the unmodified and the backdoored firmware. In both cases, we executed the IOZone write/rewrite test to create a 100 MB file with a record length of 512 KB.

Comparing the results, we can conclude that the backdoor adds an almost unnoticeable overhead to write operations. For instance, to put those results into context, we measured larger disk throughput fluctuations by changing the cable that connects the hard drive to the computer than in the case of our backdoor.

## 4.3 Data Exfiltration Backdoor

In this section, we present the design overview of a backdoor that allows to send and receive commands and data between the attacker and a malicious storage device, i.e., a Data Exfiltration Backdoor (DEB).

Basically, a DEB has two components: (*i*) a modified firmware in the target storage device and (*ii*) a protocol to leverage the modified firmware and to establish a bi-directional communication channel between the attacker and the firmware.

First we describe a concrete scenario in which the data exfiltration attack is performed, and then proceed to describe the challenges and our solution in detail.

### 4.3.1 Data Exfiltration Overview

We start with a real-world example of a server-side DEB, where the compromised drive runs behind a typical two-tier web server and database architecture, see

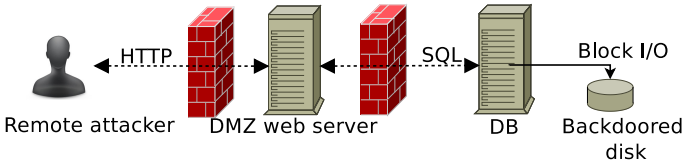


**Figure 4.4:** A server-side storage backdoor.

Figure 4.4. This scenario is of particular interest, because the various protocols and applications between the attacker and the storage device can render the establishment of a (covert) communication channel extremely difficult. We assume that the web server provides a web service where users can write and then read back content. This is the case for many web services. The specific example we select here is that of a web forum or blog service where users can post and browse comments.

To perform data exfiltration from a server, the attacker proceeds in the following way:

First, the attacker performs an HTTP GET or POST request from his or her browser to submit a new comment to the forum of the web server. The comment contains a trigger value, or *magic value*, and a disguised "read sector $X$" command for the backdoor. The web server passes this comment data and other meta-data—such as the user name and timestamp—to the back-end database through an SQL $\mathrm{INSERT}$ query. Using the filesystem and the operating system, the database then writes the data and meta-data to the compromised storage device. As one of the write requests contains the magic value, some of the comment data is now replaced by the compromised firmware with the contents of sector $X$.

Finally, the attacker issues a GET request to simply read the exact forum comment just created. This causes an SQL $\mathrm{SELECT}$ query from the web application to the database, which triggers a read request from the database to the compromised storage device. The content of the comment displayed to the attacker now contains data from sector $X$. The attacker has successfully exfiltrated data.

We stress that this DEB allows the attacker to read arbitrary sectors and access the storage device as a (remote) block device. The attacker can thus *remotely mount* filesystems and access files from the device selectively, without having to exfiltrate the storage device's contents fully.

For example, by extracting the first couple of sectors, the attacker can read the device's partition table, inferring the filesystem types in use. He or she can then follow the fileystem meta-data either locally inside the disk or remotely on his or her client machine to request individual files. We have automated this process and present results in Section 4.3.4. In conclusion, the attacker has a complete

remote read access to the hard disk.

### 4.3.2 Challenges in Implementing a DEB

While modern operating systems and disks do little to actively prevent this type of attack, we have observed some challenges that we address next.

**Data Encoding** The character encoding chosen by the application should match the one the backdoor expects. The backdoor may try different character encodings on the content of incoming write requests, looking for the magic value in the data. By knowing the encoded magic value under different encodings, the backdoor can identify which encoding is being used and encode the data to be exfiltrated such that it can be read back without conflicts by the application.

**Caching** Caching at any layer between the attacker and the storage device will cause delay, potentially both in the reception of the malicious command and the reply from the device. The delay corresponds to the time taken to evict the malicious command from caches above the storage device. Therefore, this delay depends on the load of the web server and can be influenced by the attacker.

**Magic Value Alignment** It is difficult to predict the alignment of the magic value at specific boundaries. This results in considerable overhead when searching for the magic value in a write buffer. Searching for a 4-byte magic value in a 512-byte sector, for instance, would require examining 509 byte sequences. As discussed in the next section, we mitigate this by $repeating$ the magic value multiple times in a request, such that the overhead of searching for it becomes negligible. At the same time, these repeated sequences form a suitable space for the exfiltrated data to be written to by the firmware backdoor.

While the above challenges are certainly significant and render the exploitation of the backdoor more complicated, they do not $prevent$ the use of DEBs in the general case. Our implementation provides adequate solutions to all the above complications.

### 4.3.3 Solutions Implemented

When a write request at a block number $Y$ with a to-be-written buffer $B$ is received, the backdoor checks for a $magic\ value$ in buffer $B$. In our implementation the magic value is a sequence of bytes ($magic$), and followed by a sequence of bytes ($cmd$) specifying the malicious command to be executed. As we now focus on data exfiltration, $cmd$ contains only the hex-encoded block number to be read. It would be easy to extend this encoding, for example, to support other operations, such as appending data to existing blocks, tampering

with stored data, or injecting malicious code into executables. Here, the attacker submits writes of length $2 \cdot bkdr\_bs$, formatted in the following way, with $\|$ being the concatenation operation:

$$\underbrace{magic\| \ldots \|magic}_{\text{repeated } count \text{ times}} \| \, cmd \, \| \underbrace{magic\| \ldots \|magic}_{\text{repeated } count \text{ times}} \| \, cmd$$
$$count = (bkdr\_bs - length(cmd))/length(magic)$$

Typically, there are layers (such as the filesystem) between the attacker and the disk that split all writes into blocks of at least $bkdr\_bs$ size at an arbitrary offset. Thus, the blocks created have at least one $bkdr\_bs$-sized chunk exclusively containing the repeated magic sequences followed by the command (modulo a byte-level circular permutation on the chunk, i.e., a "wrap around"). This allows the backdoor ($i$) to make sure the $bkdr\_bs$-sized chunk can be safely replaced by an equal-size exfiltrated data chunk, and ($ii$) to check efficiently for the magic value. More precisely, the backdoor checks only the first $length(cmd)+length(magic)$ bytes of the chunk, because of the possible $length(magic)$ alignments of the magic value and the possibility of the chunk starting with $cmd$. Note that increasing the length of the magic value increases the performance overhead of the backdoor. We chose a 4-byte magic value which results in a low performance overhead.

**Algorithm 1** $backdoor(blocks, magic, cmd\_size, bkdr\_bs)$

```
bkdr_count ← length(magic) + cmd_size
for blk in blocks do
  if magic present in first bkdr_count bytes of blk then
    if blk does not contain count successive magics then
      continue loop at next iteration
    end if
    cmd ← cmd_size bytes after last magic, wrap around if required
    block_num ← hex_decode(cmd)
    buf ← read_block(block_num)
    base64_encode(buf)
    blk ← buf
  end if
end for
```

If the magic value is present in $B$, the malicious behavior of the DEB is triggered: The backdoor extracts the command from the request data, such as "read data at sector $X$" for data exfiltration from the storage device, as shown in Algorithm 1. The backdoor reads data buffer $B'$ from sector $X$, encodes it using base64, which increases its size by $\frac{1}{3}$, and writes $B'$. To ensure that the encoded data

can be successfully exfiltrated, the backdoor checks for the presence of at least $bkdr\_bs * \frac{4}{3}$ bytes of consecutive magic values in a sequence of blocks and then replaces these by the base64-encoded data. At this point, a future read request at address $Y$ will return the modified content, allowing unauthorized data exfiltration of the contents at address $X$ from the device to a remote attacker.

Valid magic sequences could occur during normal, non-malicious use of the storage device. Such a false-positive would result in the storage device to detect the magic sequence and write faulty data to a sector, possibly undermining the stability of the system. However, such a false positive can only occur with negligible probability, as the backdoor always checks for about two blocks of successive magic values before attempting a replacement.

Also note that the firmware can write $B'$ to $Y$ possibly after modifications through cryptographic and steganographic operations to prevent easy detection by the administrator of the target machine.

### 4.3.4 DEB Evaluation

As we mentioned in the previous section, our backdoor in the off-the-shelf disk drive it is not stable enough to perform multiple arbitrary reading operations from the disk, which is required for implementing the complete DEB. In this section, we therefore report on experiments performed on a QEMU-based prototype.

We implemented the DEB inside QEMU's storage device functionality, which is used when using virtual IDE drives in system-virtualization software such as KVM and Xen. This provided us with an easy-to-use platform to develop, test, debug, and evaluate the backdoor.

In this case, we evaluate the data exfiltration *latency* from an attacker's point of view. In addition, we perform a file *exfiltration test* to show the feasibility of retrieving sensible remote files without needing to exfiltrate the entire disk. We base this evaluation on the scenario described in Section 4.3.1. We have conducted experiments on a virtual machine with 1 GB of memory running on a modified QEMU containing the backdoor. This is the attacker's target host. Our tests were performed on the emulated IDE disk with writeback caching. The target host runs Ubuntu and an Apache web server with two PHP scripts providing web forum (or blog) functionality. The forum shows all (recently) made comments (or "posts") using the first PHP script, and also allows the submission of new comments, using the second script. These comments are written to and read from a table in a MySQL database which runs atop an ext3 filesystem.

We emphasize here that the results of this second set of experiments highly depend on the application, the workload on the machine, and the total available system memory – and do not depend much on the caracteristics of the disk or

**Table 4.2:** Data exfiltration performance.

| | Mean (s) | 95% CI |
|---|---|---|
| Insert | 10.7 | [10.65; 10.71] |
| Latency | 9.7 | [9.55; 9.82] |
| File exfiltration | 40.0 | [39.6; 40.4] |

firmware backdoor. Indeed, because the Linux page cache[1] is essentially an LRU-like cache, forcing the eviction of pages from main memory requires generating accesses for about as much data as there is free available memory for buffers and caches on the system (and the more eager the operating system is to swap pages, the higher the memory that is available). For a single block, the time to generate that workload largely dominates the transfer time from and to the disk for a single block (even in our setup where relatively little memory is available).

We perform 30 iterations for all tests, with a 30 second pause between successive iterations. For each set of values measured, we compute 95%-confidence intervals using the t-distribution.

**Latency Test** Because of caching, the inserted comments are not immediately updated with the exfiltrated data. In fact, the malicious blocks are temporarily stored in the page cache — from where they are retrieved when they are immediately accessed by the attacker. Therefore, the presence of a cache forces the attacker to wait until the blocks are evicted from the cache. In our scenario, this can be forced by the attacker as well, namely by inserting dummy comments to quickly fill up the cache and thus force eviction of least recently accessed data.

The *insert time* in Table 4.2 shows the time taken to insert 500 8-KB comments sequentially, using the PHP form. As described in Section 4.3.3, the backdoor replaces each of these comments with 3 KB of exfiltrated data starting at the sector number included in the comment. The *latency time* in Table 4.2 shows the update latency in seconds for the 500 comments inserted during the insert test — during this time, the attacker sends many other dummy comments to speed up cache eviction. It follows that an attacker is able to exfiltrate 3000 sectors in $10.7 + 9.7 = 20.4$ seconds in our setup, achieving a read bandwidth of 74 KB/s. In practice, an attacker may limit bandwidth to avoid detection. In addition, those values will differ depending on the characteristics of the system (mainly, more physical memory will cause the comments to persist longer in cache, and more load on the server will cause the opposite). Hence, these results show that the latency is likely to be sufficiently low, and that an attacker can realistically use this technique.

[1] The page cache caches blocks read from and written to block devices, and is integrated with the filesystem cache (or buffer cache).

**Exfiltration Test** Let's now consider a typical case in which an attacker attempts to exfiltrate the `/etc/shadow` file on the target host.

To that end, we created a python program that successively (*a*) retrieves the partition table in the MBR of the disk, (*b*) retrieves the superblock of the ext3 partition, (*c*) retrieves the first block group descriptor, (*d*) retrieves the inode contents of the root directory / (always at inode number 2) in the inode table, and (*e*) retrieves the block corresponding to the root directory, therefore finding the inode number of `/etc`. By repeating the last two steps for `/etc`, the attacker retrieves the `/etc/shadow` file on the target host.

Table 4.2, row 3, shows that `/etc/shadow` can be exfiltrated in less than a minute. Because the process of retrieving the file requires nine queries for a few sectors, each of them depending on the results returned by the preceding query, this figure is mainly dominated by the time taken to evict comments from the cache. This means that the actual latency for a single sector is about 4 seconds (for a comparison, note that the latency figure in row 2 also includes the retrieval time of the 3000 sectors).

## 4.4 Detection and Prevention

We first discuss the applicability of existing standard techniques for defeating or mitigating DEBs, including encryption of data at rest, signed firmware updates, and intrusion detection systems. Subsequently, we propose two new techniques specifically targeting the detection of DEBs: OS page cache integrity checks and firmware integrity verification.

### 4.4.1 Encryption of Data at Rest

The use of encryption of data at rest is still an exception, both on servers and desktop computers. When used, it is often for the purpose of regulatory compliance or to provide easy storage-device disposal and theft protection (by securely deleting the encryption key associated with a lost disk). *Under some conditions*, encryption of data at rest mitigates the possibility of data-exfiltration backdoors on storage devices: it renders establishing a covert communication channel more difficult for remote attackers and prevents the untrusted storage device from accessing the data in the first place.

**Hardware-Based Disk Encryption** Hardware-based disk-encryption mechanisms commonly rely on the hard disk drive to encrypt data itself. Decryption is only possible after a correct password has been provided to the drive. In such a setup, as data is encrypted and decrypted within the drive, a backdoor would only

have to hook into the firmware before the encryption component. Thereafter, the hard-disk will encrypt and decrypt data for the backdoor.

**Software-Based (Filesystem and Partition) Encryption** Other hard-disk encryption systems, among them BitLocker, FileVault, and TrueCrypt, encrypt full partitions over arbitrary storage devices. Such mechanisms often rely on a minimal system to be loaded from a non-encrypted partition whose integrity is verified by a trusted boot mechanism. A trusted boot mechanism relies on a TPM to prevent a modified system, e.g., modified by the drive itself, to access a protected key sealed by the TPM. However, without an IOMMU, the backdoor on a hard drive can launch a DMA attack [Dor04] to read arbitrary locations from the main memory. This allows the backdoored hard disk to obtain the encryption key. Recently, it has been shown that even mechanisms to protect encryption keys against DMA attacks [MFD11] can be circumvented [BR12].

In conclusion, neither hardware-based nor software-based encryption offer full protection against DEBs in all cases. Disk encryption *can* prevent DEBs as presented in this chapter when keys are not managed by the disk itself and when the disk is not able to use DMA to access main memory. This corresponds to setups in which:

- system-level encryption is used *and* disks are attached to the computer (e.g., desktops or laptops) *and* an IOMMU (e.g., Intel VT-d or AMD SVM) is present and properly configured;
- system-level encryption *and* remote storage are used, for example, servers with a Network Attached Storage (NAS) or Storage Area Network (SAN). Such a remote storage must not support remote DMA capabilities, like Infiniband or Myrinet protocols does.

We believe that both setups are uncommon. While IOMMUs are present in many computers, they are rarely activated because of their significant performance overhead [BYXO+07]. On the other hand, servers that rely on a SAN or NAS are typically not using software disk encryption because of its significant performance impact.

### 4.4.2 Signed Firmware Updates

To protect a device from malicious firmware updates, cryptographic integrity checks can be used. The use of asymmetric signatures is preferable in this case, and each device would be manufactured with the public key of the entity

performing the firmware updates. Although the idea of signing the firmware is widely known, we have not been able to assess how widespread its use is for hard-disks and storage devices in general. We have found evidence that some RAID controllers [RSA13] and USB flash storage sticks [Kin13] have digitally signed firmware, but these appear to be exceptions rather than the rule.

Nevertheless, signed firmwares do not prevent an attacker with physical access to the device from replacing it with an apparently similar, but in reality backdoored, device. Also note that the recent compromise of certification authorities, software vendors' certificates, and hash collisions has demonstrated real-world limitations of signature mechanisms.

Finally, firmware signatures merely check code integrity at load time and do not prevent modifications at run time. A vulnerability in the firmware that is exploitable from the ATA bus[2] would allow infection of the drive, bypassing the signed update mechanism. In addition, such vulnerabilities are likely to be easily exploitable, because no modern exploit-mitigation techniques are present in the disk firmwares we analyzed.

### 4.4.3 Intrusion Detection Systems

Current network-based intrusion detection systems and antivirus software products use, to a large extent, simple pattern matching to detect known malicious content. The DEBs presented in this paper could be detected by such tools if the magic value is known to the latter. This can be the case if the attacker targets a large number of machines with the same magic value, but is inadequate for targeted attacks. For instance, an attacker could change the magic value for each target machine or it would be possible to make the magic values a time-dependent function to evade detection. Finally, the attacker's channel used for communication with the firmware may be encrypted. We conclude that today's intrusion detection systems do not offer a strong protection mechanism against DEBs.

### 4.4.4 Page-cache-driven Integrity Checks

In addition to the standard mechanisms presented above, one could also envision detection technique that relies on the page cache. Most filesystems leverage the page cache to significantly speed up workloads by caching most recently accessed blocks. We propose to modify the page cache to also perform probabilistic detection of DEBs. As the cache contains recently written data, it can be used to check the integrity of disk-provided data.

[2] Or an insecure functionality that could be abused without physical access.

More precisely, the cache would allocate a new entry on write misses, and, after the data has been written to the disk (immediately for write-through caches, and after laundering for writeback caches), subsequent reads from the cache would be randomly subject to asynchronous integrity checks. The checks would simply read back data from the disk and check for a match.

However, with deterministic cache-eviction algorithms such as *least recently used* (LRU), both the disk and the remote attacker could estimate the size of the cache in use, and the attacker could adjust queries to guarantee that the data has been evicted from the cache by the time it is read back. Therefore, we suggest to partially randomize the cache eviction policy. For instance, a good candidate would be a *randomization-modified LRU-2* algorithm, whereby the eviction from the first-level cache to the second-level cache would remain LRU, but the eviction from the second level cache would be uniformly random. This technique would introduce a performance overhead, but we conjecture that this could be an acceptable trade-off for detecting such backdoors in the wild.

### 4.4.5 Detection Using Firmware Integrity Verification

Recent research in device attestation [LMP11] could be applied to detect malicious firmwares. However, we note that device attestation is controversial [CFPS09], especially in the specific context of this work: the firmware is typically stored in different regions of the drive (such as disk platters and serial flash), and accessing those different regions is slow and subject to various time delays. Delays are difficult to predict, and this questions standard assumptions made by existing software-based attestation techniques, rendering them ineffective in our scenario.

However, one could leverage the fact that the disk always starts executing from the ROM code, essentially providing a hardware root of trust. By interfacing with the ROM bootloader and using it to control execution and verify code loading one could guarantee that only correct code was loaded.

# Chapter 5

# Towards automating platform reverse engineering of embedded devices

## 5.1 Introduction

*Avatar* has proven to be a useful tool for analyzing firmware which is tightly coupled with its hardware platform. Thanks to *Avatar*, one needs significantly less prior knowledge of the hardware platform to perform analyses of particular regions of interest in embedded code. However, the framework has some limitations when applied to whole-system execution.

We identified three major issues when forwarding all I/O accesses with *Avatar* to physical hardware. First, using physical hardware makes it much harder to control input. Hence, experiments are less deterministic, as a rerun might have different timer values, different sensor readings, and so on. Second, we faced several problems caused by execution time in *Avatar* being much slower than before. One such problem occurred with the DRAM controller initialization code of the HDD which we reversed in Chapter 4. Apparently, the controller's registers need to be configured within a certain timespan, which was exceeded when the code was emulated. Thus we had to execute this code snippet on the device to initialize the controller properly. Another case was the reading of hardware timers. Here, the firmware reads the timer repeatedly and checks that the difference between readings is small, most certainly to prevent jitter. Third, symbolic execution with *Avatar* is tricky when physical hardware is accessed. Once a value has been written to a hardware device in one symbolic state, the device's state is not any more coherent with all other symbolic states. If the device was emulated, its internal state could be kept synchronized with symbolic states.

To summarize, *Avatar* would benefit from moving peripherals from the physical device to the emulator. However, the framework was designed in the first place so that one would not have to reverse-engineer the hardware platform, which is in general a difficult and time-consuming work.

Embedded systems are typically designed as a System on Chip (SoC), which means that the processor core and peripherals are located on the same silicon. Peripherals communicate with the processor through buses (e.g., Advanced Peripheral Bus (APB), AMBA High-performance Bus (AHB) and Advanced eXtensible Interface (AXI), different versions of ARM's Advanced Microcontroller Bus Achitecture (AMBA) bus). Further, peripherals may have one or several clock inputs, and can connect directly to pins of the processor, the interrupt controller, and the Direct Memory Access (DMA) controller.

Most bigger chip vendors have one or several standard platforms, with a default peripheral for each task. Examples are Texas Instruments (TI)'s OMAP platform, Xilinx' Zynq platform, etc. However, manufacturers are also free to integrate hardware blocks (IP cores) from other companies into their own ASIC. An ASIC is specifically built for one purpose (like a hard disk controller), and does not necessarily conform to other platforms.

Of course, designing a non-standard system requires adapting firmware. Embedded systems, unlike Personal Computers, do not have a standardized system for hardware discovery. PC based systems today provide auto-discovery mechanisms like the Advanced Configuration and Power Interface (ACPI) and the Peripheral Component Interconnection (PCI) bus to assist the operating system in finding devices and choosing appropriate drivers. However, in most embedded systems, knowledge about peripheral devices in a SoC are usually encoded only in the firmware. As a consequence, firmware code has implicit expectations about the platform it is running on, and needs to be adapted to each platform. While this is easy if the Software Development Kit (SDK) and all IP cores used are from one company, significant work is needed to integrate drivers for peripherals from other companies.

Software like *Das U-Boot* (a popular open-source bootloader) and the *Linux kernel* aid in this task by providing a library of drivers for most peripherals. The platform layout is described in a static data structure, called *"Device Tree"* [pow11, ope94, GH06] (also known as *Flattened Device Tree (FDT)* or *Device Tree Blob (DTB)*). The *device tree* is a tree-like structure, where inner tree nodes are bus controllers and leaf nodes are devices. Each node contains key-value assignments, called attributes, describing the hardware more closely. More complex peripheral relationships, like interrupts, can be expressed with pointers to other nodes (called handles, which effectively transform the tree structure into a directed graph). Both a human-readable textual representation (an example is given in Listing 5.1) and a binary representation suitable for passing a *device tree* to firmware have been defined. The specification detail of device tree is

sufficient to instantiate peripherals in an emulator, as we will demonstrate in Section 5.4.

In this chapter, we propose a method to fingerprint peripherals. We first capture traces of interactions between the firmware and some emulated peripherals with $S^2E$. Then, we create fingerprints of peripherals based on their interactions. These fingerprints of known, emulated devices are stored in a database. When a fingerprint is now obtained from traces of another peripheral, it can be compared against the database to get likely matches. In the end, we build a platform description with sufficient detail to instantiate an emulated version of the embedded device from these matches.

**Listing 5.1:** An example of a device tree.

```
/* An example of the textual device tree representation */
/ {
    model = "ARM Integrator/CP";  /* A human-readable board name */
    /* The "compatible" attribute specifies a list of machine-readable names
       that this board is compatible with (in decreasing order) */
    compatible = "arm,integratorcp";
    /* All child nodes use one cell (integer value) to represent addresses */
    #address-cells = <1>;
    #size-cells = <1>; /* And one cell for sizes */

    /* The board's memory (SRAM, DRAM) is registered here */
    memory@00000000 {
        device_type = "memory";
        /* "reg" attributes contain a list of address, size pairs.
           The number of cells used for address and size depends on the
           #address-cells, #size-cells attribute of the closest parent */
        reg = <0x00000000 0x08000000>;
    };

    cpus {
        /* CPUs are just numbered, not mapped to the global address space */
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            device_type = "cpu";
            compatible = "arm,arm1136";
        };
    };

    /* This is the interrupt controller */
    pic0x14000000 {
        compatible = "arm,versatile-fpga-irq";
        /* Interrupt pins on this controller are specified with one cell */
        #interrupt-cells = <1>;
        interrupt-controller;
        reg = <0x14000000 0x100>;
```

```
        /* Some attribute specific to the Linux driver of this device */
        clear-mask = <0xffffffff>;
    };

    /* This is a bus */
    fpga {
        /* The "ranges" attribute can be used to remap the bus to another
           memory location. Here no value is specified, so the bus is just
           referring to the global address space. */
        ranges;
        compatible = "arm,amba-bus", "simple-bus";
        /* Device on this bus send interrupt requests to this device (pic) */
        interrupt-parent = <&pic>;

        /* A serial port */
        uart@16000000 = {
            compatible = "arm,pl011", "arm,primecell";
            reg = <0x16000000 0x1000>;
            /* The serial port's interrupt is connected to pin 1 of the
               interrupt controller */
            interrupts = <1>;
        };
    };
};
```

## Problem statement and contributions

*Avatar*'s idea of forwarding I/O interactions to the physical platform while executing the firmware in an emulator helps to avoid tedious reverse engineering of the embedded platform. However, this method has some shortcomings in execution time fidelity and incompatibilities of concrete physical and emulated symbolic domains in symbolic execution.

Hence, we propose a method to use the current *Avatar* system to automatically fingerprint embedded device peripherals, and suggest similar, known peripherals from a fingerprint database. The goal of our system is to create an initial assumption of the embedded platform's device description, which can then be adjusted and used to instantiate an emulator for the device.

Further, we discuss issues with systems where some peripherals have been moved to the emulator domain, and some remain on the physical device, and suggest strategies to mitigate those.

## Scope

In this work, we focus on detecting Memory-Mapped Input-Output (MMIO) devices. There are other ways to connect peripheral devices, like dedicated I/O

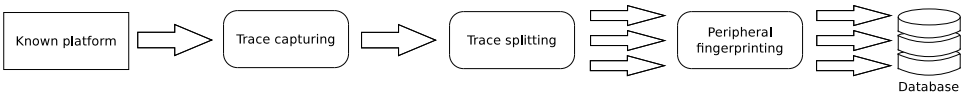


**Figure 5.1:** Steps for training the peripheral identification system.

address spaces (in the x86 architecture), or dedicated processor instructions such as the coprocessor interface of ARM processors. However, most embedded device architectures use memory-mapped I/O to expose peripherals to firmware.

Besides, understanding direct connections between peripherals is out of scope of this work. Such a connection can be an interrupt line which is directly connected to the interrupt controller, or a DMA request line connected to the DMA controller. In Section 5.5, we discuss how identifying interrupt controllers and DMA controller can help in building a model of those which will be able to uncover these connections.

### Overview of the chapter

In Section 5.2, we describe the design of our system and specific design choices we made. Section 5.3 then details how the design was implemented on top of *Avatar*. Subsequently, in Section 5.4, we evaluate the system on several embedded platforms and draw conclusions from its performance. Finally, we discuss some implications of mixing peripherals on a physical device and in the emulator in Section 5.5.

## 5.2 Methodology

In this section, we present the design of our embedded peripheral identification system. Peripheral identification is split into two parts: First, in a learning phase (see Figure 5.1), a database of labeled peripheral device fingerprints is built. Then, in a second phase (see Figure 5.2), fingerprints of unknown peripherals are compared against this database to identify similarities to known models. Both phases share the initial steps of capturing MMIO traces of the communication between firmware and peripherals, breaking those traces into individual traces per peripheral, and aggregating information into peripheral fingerprints. The identification phase additionally includes a comparison of generated fingerprints against the database. We describe each of these steps in more detail in the following subsections.

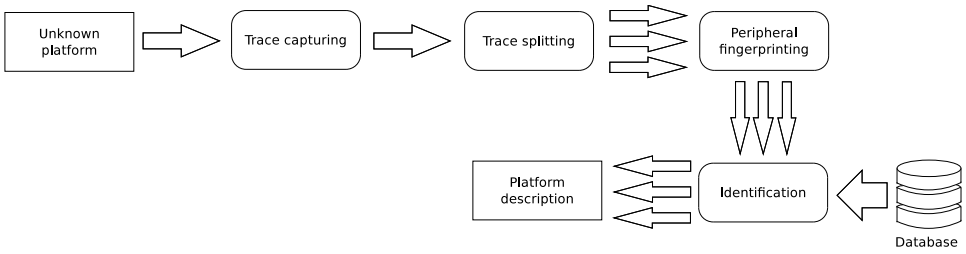


**Figure 5.2:** Steps for identifying unknown peripherals.

### 5.2.1 Gathering traces

In a first step, we collect MMIO traces of one or several peripherals of an embedded system. An important aspect of trace acquisition is the proper exercising of the device from which traces are collected. The more functionality of the device is explored during this phase, the better the device's fingerprint will be afterwards. On the serial port, for example, it would be desirable to see MMIO operation as well as DMA operation, changing of baud rate and other line parameters, and data input as well as output. Obviously, the exercising phase is highly dependent on the device class.

We considered several approaches, both hardware-based and emulator-based, to collect traces. As our goal is to build emulators for identified hardware platforms, we need to fingerprint emulated peripherals first to build the fingerprint database.

The output of this step is a memory trace containing MMIO accesses with the following information: physical address, access size, value, program counter at access, CPU state register at access.

### 5.2.2 Splitting traces per peripheral

Once traces have been obtained, they need to be broken down to individual peripherals. Each peripheral occupies one range in the global address space. [1] We then assume a minimum memory region size. Choosing a very small value will result in many fragmented devices being detected, which should in fact be a single device. On the other hand, choosing a very big initial value will group several different devices together, which should in fact be single devices.

The best value we found as smallest memory region assumption was 0x100 bytes. Even though there are some devices which have smaller memory region sizes, the risk of grouping them together is low. Due to memory alignment constraints, address space between peripherals is often left unused. Taking this unused space into account, 88% of peripherals in the Linux device tree specifications were

[1] There are very few peripherals, like a PCI bus interface, that can have several associated memory ranges. In a future version we plan to add handling for those peripherals as well.

located at least 0x100 bytes apart. Furthermore, the understanding of what is "one peripheral" differs considerably between Qemu and the Linux kernel. For instance, Qemu tends to group "platform devices" (like several timers) together, where Linux does not. In fact, most devices in the Linux device tree specifications located less than 0x100 bytes apart are of this kind.

The address space is then divided evenly into bins of the minimum peripheral memory range size, and all accesses from the trace are grouped into their respective bin. Subsequently, adjacent bins are merged if they are believed to belong to the same peripheral. The decision to merge two bins is made based on a linear distance function. This function maps the base address distance between two bins and the minimum code distance of memory access locations in the bin. While the motivation for taking the base address of bins as parameter is evident, the code distance is less obvious. The rationale behind this parameter is that driver code for one peripheral is contained in one or a small set of source code files. A linker will most likely place generated binary object files close together in the final binary, which is why we expect memory accesses from close code locations to belong more likely to the same peripherals than accesses from code locations which are far apart. Of course, code inlining breaks this assumption, as code locations accessing the device will be distributed across the firmware. Hence, we chose to take only the minimum distance between code locations of two neighboring bins into account, thus increasing the probability that at least one access location in the bin is made from non-inlined driver code.

The threshold used in the distance function was empirically determined to minimize the errors in our experiments.

### 5.2.3 Fingerprinting a peripheral

Once the memory trace has been split by peripheral, we generate a fingerprint from each peripheral's accesses. Our fingerprint is inspired by register descriptions like they are commonly found in datasheets. Each register is described by several features, for example the register's direction (read, write, or read/write), its width and access frequency. The following list explains all collected features in detail.

- **Register size** as used by the code to access the register.
- Estimated **real register size**. Due to alignment requirements, some platforms use larger integer types to access device registers than needed. For example, the popular UART 16550 design has byte-sized registers, but was found to be used with word-sized (4 byte) accesses in a platform. We identify the effective size by testing if the value can be stored in a one, two, four or eight byte integer without loss of information. This feature

enabled the machine learning algorithm to identify two register models as the same even though different access sizes were used.

- **Access frequency** as a fraction of total accesses to all of the device's registers. This value helps to distinguish configuration registers from flag and data registers, as configuration registers should be initialized only very rarely (during device initialization and configuration changes).

- **Register direction**. A register can be either read-only (when it was read from but never has been written to), write-only (when it was written to but never has been read from), or read-write.

- **Bit vector detection**. If a register value is always manipulated with logical operations (and, or, shift) just before being stored or just after being read, it is likely to contain bit flags. This feature counts the frequency of accesses preceded or followed by logic operations, and is designed to identify flag and configuration registers organized as bit vectors.

- **Mean and variance of value difference**. The distribution of the difference between two consecutive accesses. This feature is designed to discover increments registers which increase or decrease continuously (e.g., a timer's data register).

- **Mean and variance of the Hamming distance**. Similar to the previous feature, we observe the distribution of the number of changed bits between two consecutive accesses. Flag registers, for example, only change a few bits at a time. We expect the Hamming distance (number of changed bits) to be a good measurement to detect such behavior.

- **Value mean and variance**. This feature is designed to measure the distribution of values written to and read from this register. While it is dangerous to draw conclusions on data from higher protocol layers, our expectation is that we can exercise the device well enough so that the machine learning algorithm will only use these values when they are pertinent to the peripheral.

- Accessed in **interrupt mode**. The frequency at which register accesses happen while the CPU is in interrupt mode. A peripheral's interrupt flag is usually accessed in this mode, and should thus be detectable through this feature.

To avoid biased peripheral descriptions based on some specific behavior of one firmware or platform, several normalized access traces of the same peripheral from different platforms and/or running with different firmware can be merged before the fingerprinting phase.

### 5.2.4 Identifying similar fingerprints

In the final step of our approach, we study several classifiers to identify a fingerprint of an unknown peripheral from the gathered information. We test machine learning approaches as well as a manually designed fingerprint comparison function and evaluate their performance. In order to train the machine learning algorithm, we need a labeled dataset of peripherals' fingerprints. Fortunately, the Linux kernel already contains a large database of well-identified device peripherals in the form of device tree files. Thus, we can simply label peripherals for most of the used devices by using these device tree descriptions. It is much harder to find the peripheral design that a concrete peripheral is based on for devices where no such description is available. Most of the time, the manufacturer provides a description of the registers which we can manually check against the automatically extracted register model. However, he will not tell the name and brand of the IP block the peripheral design is based on (i.e., the information conveyed by the device tree *"compatible"* attribute). In these cases, we simply verified manually that the extracted register model conforms to the datasheet, and that the behavior of suggested similar peripherals is compatible with the datasheet.

## 5.3 Implementation

In the previous section, we described the abstract design of our peripheral identification system. Here, we will outline specific implementation decisions for this design. First, we discuss several methods for tracing memory accesses with hardware methods and emulation. Then, trace analysis is described briefly, highlighting only points which are not evident from the fingerprint description in the methodology. Finally, a subsection is dedicated to the fingerprint comparison.

### 5.3.1 Trace recording

We considered and evaluated several options for recording MMIO access traces to varying degrees. The remainder of this subsection is dedicated to first a description of hardware-based tracing methods, and then emulation-based memory access recording.

#### Synthesizing peripherals on an FPGA

Field Programmable Gate Arrays (FPGAs) are a fast way of testing chip designs specified in a hardware design language (e.g., VHDL). Many IP cores of embedded peripherals which are synthesizable on an FPGA are available in open-source

public libraries, for example LEON/GRLIB[man15] and OpenCores [webd]. We considered supplementing these designs with a bus tracer [YLKH11] to record peripheral accesses. Using a development board which can run Linux and synthesize designs on an FPGA, we could quickly program the FPGA with peripheral IP blocks. The Zedboard, an evaluation board based on the Xilinx Zynq family with a dual-core ARM Cortex A9 core as well as an FPGA, fulfills this requirement. On the ARM core, we could then run a Linux kernel compiled with drivers for the peripheral to exercise the device and generate traces.

Eventually we decided against this solution, as the steep learning curve for FPGA programming, choice of the right peripherals from libraries, and getting those devices to work in Linux seemed to harbor too many pitfalls. However, we want to study this solution more in the future, as this experimental setting seems most suited for tracing a large set of peripherals.

### Using hardware debug and trace technology

A very straightforward idea would be to use ARM Embedded Trace Macrocell (ETM) and Data Watchpoint and Trace (DWT) technology to generate instruction and memory access traces of platforms in real-time. This option was not viable for us, as some of the evaluation boards at hand were lacking the ETM core. Others had the necessary hardware (e.g., the BeagleBoard), but were not compatible with our tools As for the previous technique, we would like to explore this solution in more depth in future work.

### Dynamically creating emulated platforms

The concept of an emulator where platforms can be configured dynamically seems perfectly suited for our problem. Here, we could configure a minimum platform, and then add one peripheral at a time. A Linux kernel including drivers for the peripheral could then do the exercising. There was already a version of Qemu in the Xilinx Zynq SDK capable of adding peripherals [webg]. We extended this version to work with arbitrary peripherals in the expectation that we could use device tree specifications from the Linux kernel to generate an emulator, and then run the Linux kernel inside this emulator. Unfortunately, it turned out there are several limitations in this approach. First, not all the peripherals specified in the device trees are emulated by Qemu. Thus it is necessary to pass Linux a device tree which contains only the actually emulated devices. Second, embedding peripherals into another platform's device tree specification is non-trivial. Different device trees have subtle differences in vocabulary. For example, a peripheral's interrupt connection can be specified with the *"interrupts"* and *"interrupt-parent"* attributes, or with the *"interrupts-extended"* attribute. Further, each device can have special attributes which need to be taken into consideration in the emulation (e.g., some serial ports use the *"reg-shift"* parameter to specify a left-shift

to be applied to register offsets). Finally, addressing and naming conventions vary across boards. On the Zynq boards, interrupt pins are addressed with three cells, while ARM boards use only one cell to number interrupt pins. One board uses clocks with unnamed pins, but a serial driver from another board makes assumptions about the name of clock pins and fails the initialization if the name cannot be resolved.

All together, these issues required too much effort to fix by hand every single device tree specification and adapt it to our purposes. As a result, also this solution was infeasible for our experiments.

### Using existing emulated platforms

Finally, we settled on using platforms already supported by Qemu/S$^2$E, and ported emulation of the TI/Omap3 platform from Linaro's Qemu branch. As most of the platforms available in Qemu are quite old, obtaining working operating system images posed some challenges. Some boards were still supported by U-Boot and Linux, in which case we compiled binaries with recent code bases. Already-assembled firmware images were available on the Internet for others cases and, finally, some boards required to use specific versions of OpenEmbedded [webe] and Poky Linux [webf]. The emulated boards used in this work are:

- **integratorcp**, based on the ARM Integrator platform, with U-Boot and Linux
- **versatilepb**, based on the ARM Versatile platform, with U-Boot and Linux
- **beagle**, an emulation of the BeagleBoard based on the TI Omap3 platform, with U-Boot
- **smdkc210**, an emulation of a development board for the Samsung Exynos platform, with Linux
- **xilinx-zynq-a9**, based on the Xilinx Zynq platform, with U-Boot and Linux
- **n800**, an emulation of a Nokia N800 navigator based on the TI Omap3 platform, with Linux
- **connex**, an emulation of a Gumstix Connex board, based on the Intel/-Marvell PXA 255 architecture, with U-Boot and Linux
- **verdex**, and emulation of a Gumstix Verdex board, based on the Intel/-Marvell PXA 270 architecture, with U-Boot and Linux

We ran each board in S$^2$E with a configuration to trace memory accesses to any peripheral (peripheral ranges were identified by reading Linux device tree specifications and the Qemu source code) and the start of execution of each translation block. The existing *InstructionTracer* plugin in S$^2$E was enhanced to log the LLVM name of each translation block. Similarly, the *MemoryTracer* plugin was adapted to monitor virtual or physical memory ranges, and to log physical addresses. Finally, a new plugin, *DumpLLVMBitcode*, was introduced to store the generated LLVM bitcode at the end of the emulation.

### 5.3.2 Trace analysis

Trace analysis is split across several different programs. First, a C++ program processes the huge execution trace files, and extracts information on memory accesses into a JSON (a simple object serialization format) file. Generated LLVM bitcode is processed by another program to extract information on memory load and store instructions into another JSON file. The LLVM code generated by S$^2$E always reads and stores register values to Qemu's central CPU structure (`CPUARMState`), which makes data flow analysis more difficult. Thus, the bitcode is first transformed to eliminate consecutive store - load sequences for registers in the same basic block. It then checks for any bitwise operation in the same basic block on the data flow upgraph (for reads) or downgraph (for writes).

These two file are then consumed by a python script, which generates a fingerprint. All values are aggregated per peripheral and register as described in Section 5.2.3.

### 5.3.3 Machine learning of fingerprints

To be usable as input for a machine learning algorithm, each device must have a fixed vector of features. As devices can have register sets of different lengths, we limit the identification to the first twenty registers. The register index is calculated by dividing the register offset with the greatest common divisor of all register offsets of one peripheral. This measure is taken to ensure that devices with different architecture-dependent access sizes, which are otherwise identical, are recognized as the same. Then, all register indices from 0 to 19 are iterated. In case a register is not present in the register map, all its features are assumed to be zero. The gathered feature vectors of each register are concatenated, yielding a vector of 12 features per register times 20 registers = 240 features for a device fingerprint.

The fingerprints are then labeled with the device tree "compatible" attribute. We use the Weka machine learning framework [HFH+09] to classify the data.

Our manually designed fingerprint comparison borrows on the concepts of NMap's operating system fingerprints [Lyo09]. The fingerprint is discretized further, re-

ducing per-register information to the register's direction, maximum value size in bytes, access frequency ("frequently", "sometimes", "rarely" accessed, or "unknown") and data type ("bit vector", "integer" or "unknown"). A comparison of two fingerprints then boils down to a comparison of registers. Registers are referenced by their index, i.e., their offset divided by the peripheral's access size. This measure insures that same peripherals with different access sizes are still identified as being the same. For each register present in both fingerprints, a full point is awarded if the discretized information matches perfectly, and half a point if the information is one category off (e.g., if the access frequency is "sometimes" instead of "frequently"). Features where one of the fingerprints has an "unknown" value are ignored in the comparison.

Finally, the register score is normalized to one, and added to the total similarity value. This value is again normalized, where registers present in both fingerprints are given a weight of one, and registers present in only one peripheral have a weight of a half. Thus, the final value is between 1.0, which is a perfect match, and 0.0, which means that the two peripherals do not share any common registers.

## 5.4 Evaluation

In this section, we evaluate the performance of our identification system. The work in this section is ongoing research, we expect to improve the shown results in the future with a more carefully curated dataset and the acquisition of additional traces.

Our dataset currently contains 66 traces for 34 peripherals. To evaluate cross-platform detection of peripherals, at least two platforms with the same peripheral are needed, e.g., the **integeratorcp** and the **versatilepb** platform share the same devices. Further, different firmware needs to be run on the same platform (e.g., **u-boot** and **linux**) to evaluate if a reliable fingerprint can be extracted from different firmware behaviors. An overview of the different peripherals and the number of traces is given in Table 5.1.

First, we present a machine learning approach on the extracted fingerprints, and show that these methods are not adapted to the small number of samples available. Then we perform an exemplary manual comparison of the extracted register information against a product datasheet. Finally we describe a more successful manual fingerprint comparison heuristics.

### 5.4.1 Supervised classification and K-Means clustering

Due to the low number of samples, supervised classification algorithms like decision trees are hard to train and evaluate properly. Most peripherals are repre-

| peripheral device | number of traces |
|---|---|
| arm,core-module-versatile | 1 |
| arm,integrator-cp-syscon | 1 |
| arm,pl031 | 1 |
| arm,pl041 | 1 |
| arm,sctl | 1 |
| arm,versatile-sic | 1 |
| samsung,exynos4210-pmu | 1 |
| ti,omap3430-timer | 1 |
| ti,omap3-i2c | 1 |
| ti,omap3-prm | 1 |
| ti,omap3-scrm | 1 |
| unknown2 | 1 |
| unknown3 | 1 |
| unknown4 | 1 |
| unkown1 | 1 |
| xlnx,ps7-nand-1.00.a | 1 |
| arm,pl050 | 2 |
| arm,pl180 | 2 |
| arm,versatile-fpga-irq | 2 |
| marvell,pxa-intc | 2 |
| mrvl,pxa-lcdc | 2 |
| mrvl,pxa-mmc | 2 |
| mrvl,pxa-timers | 2 |
| pxa,coremodule | 2 |
| pxa,mm | 2 |
| smsc,lan91c111 | 2 |
| xlnx,ps7-uart-1.00.a | 2 |
| arm,core-module-integrator | 3 |
| mrvl,pxa-gpio | 3 |
| arm,gic | 4 |
| arm,integrator-cp-timer | 4 |
| arm,pl011 | 7 |
| mrvl,pxa-uart | 7 |

**Table 5.1:** Number of traces per peripheral.

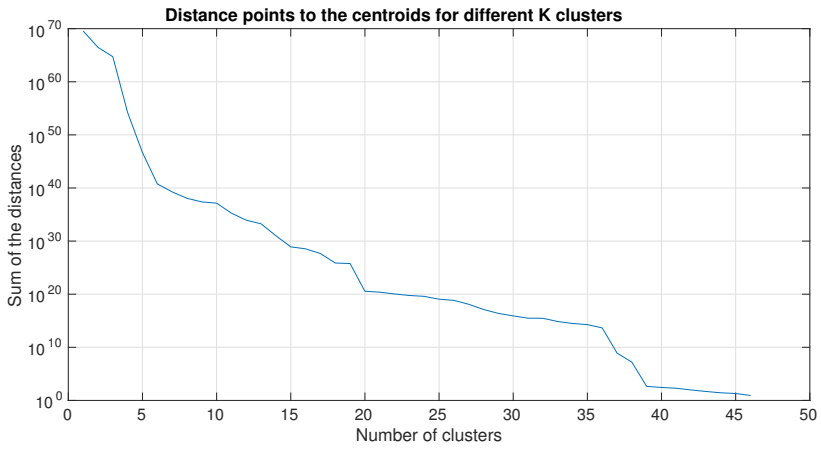


**Figure 5.3:** K-Means clustering of peripheral fingerprints.

sented by only one or two samples. In this case, machine learning cannot find a classifier without underfitting or overfitting, and is simply not the right tool for the task.

Another way of analyzing high-dimensional data with few samples is k-means clustering. We plotted the sum of distances of all points from their centroid in Figure 5.3. At 20 clusters, a drop is clearly visible. As some traces, for example from secondary and tertiary serial ports, contain very little information, the denser clusters here conform to our intuition. After 35 clusters, one cluster per device exists, and the clusters become again visibly denser.

### 5.4.2 Evaluation against datasheet

Exemplary, we evaluate a trace of a **pl011** serial port on the **integratorcp** board with **u-boot**. Of course, this showcase evaluation cannot replace a more rigorous one. The idea is to demonstrate that different learning methods might be more promising with the current dataset.

In Table 5.2, three registers can be seen. According to the datasheet, register 0 is the data register. To output data, the driver simply writes to this register. Similarly, to read data from the port, the driver reads this register. These properties are well reflected in the data direction (rw), and the access frequency (very frequent). Further, register 6 is a flag register. It signals when data has been sent, and when data has been received. One can see from the high bit vector frequency (0.71) and the Hamming distance distribution that this register contains a bit vector. The data direction is read-only. Finally, register 9 is a configuration register. It is written only once, and never read.

| Index | 0 | 6 | 9 |
|---|---|---|---|
| Offset | 0 | 24 | 36 |
| Size | 4 | 4 | 4 |
| Actual size | 1 | 1 | 1 |
| Direction | rw | r | w |
| Bit vector frequency | 0.02 | 0.71 | 0.0 |
| IRQ mode frequency | 0.0 | 0.0 | 0.0 |
| Access frequency | 0.41 | 0.59 | 0.00 |
| Value distribution | $\mu = 82.97$, $\sigma = 36.05$ | $\mu = 142.70$, $\sigma = 19.60$ | $\mu = 24.00$, $\sigma = 0.0$ |
| Difference distribution | $\mu = 0.01$, $\sigma = 39.85$ | $\mu = -0.01$, $\sigma = 3.55$ | $\mu = 0$, $\sigma = 0$ |
| Hamming distance distribution | $\mu = 2.64$, $\sigma = 1.32$ | $\mu = 0.00$, $\sigma = 0.00$ | $\mu = 0$, $\sigma = 0$ |

**Table 5.2:** An excerpt of the pl011 uart's register map.

| peripheral | fingerprint 1 | fingerprint 2 | similarity |
|---|---|---|---|
| pl011 | integratorcp - u-boot | integratorcp - linux | 0.80 |
| | integratorcp - linux | versatilepb - u-boot | 0.80 |
| arm,timer | integratorcp - u-boot | integratorcp - linux | 0.95 |
| | integratorcp - u-boot | connex - linux | 0.78 |
| | integratorcp - linux | connex - u-boot | 0.83 |
| smsc,lan91c111 | integratorcp - u-boot | integratorcp - linux | 0.81 |

**Table 5.3:** Peripherals identified by heuristics

### 5.4.3 Manual fingerprint comparison heuristics

Here we evaluate the performance of the manually-designed fingerprint comparison heuristics. As can be seen from Table 5.3, the heuristics successfully identifies three different peripheral devices across three different platforms and two different firmware. With the current cutoff threshold (0.75), no false positive detections are present. Below this threshold, false positives will start to appear, as some devices have very similar register access patters. For example, ARM primecell devices all have a common register block for device identification. These registers are accessed by Linux, but then the device is not initialized further, making different devices look alike from their access patterns.

It is also interesting to note that the peripheral detected as a timer in the **connex** platform is labeled as core module in the device tree file. Further investigation showed that the core module actually incorporates the identified timer hardware.

While the detection heuristics is surely can be optimized more, e.g., by adding more features to the comparison and by better optimizing the weights of indi-

vidual features, this proof-of-concept implementation is already useful. Since the number of false positives is low, the current system can assist in reverse engineering, and suggest highly likely similar peripherals to the reverse engineer.

### 5.4.4 Instantiating an emulator from a reverse-engineered device tree file

We manually identified peripheral devices on the integratorcp platform from traces of **u-boot** running. From there, we could create a minimal device tree file, which was then used to emulate the board again. To this end, we modified Xilinx' Qemu version to also support the creation of other boards beside the Zynq platform from device tree files. An emulator instantiated from a reconstructed device tree file for the integratorcp and versatilepb platform was able to run **u-boot**.

## 5.5 Conclusion

In this section, we shortly discuss some difficulties in running platforms on Avatar in a mixed mode, where some some peripherals are emulated and others are on the physical device. Then we conclude this chapter with a review of the work.

### 5.5.1 Considerations in mixed emulated and physical execution

Having some peripherals located on the physical device and some peripherals in the emulator brings some new challenges. Here, we are going to highlight those challenges which need to be addressed to achieve mixed emulated and physical execution.

For very simple cases, mixed execution requires no additional effort. We were able to simply move the serial port of the hard disk (see Chapter 4) to the emulator while forwarding all other peripheral accesses to the physical platform with *Avatar*. However, whenever there are interconnections between peripherals, move one from the physical platform to the emulator gets more tricky.

Let us assume that a hardware timer has been moved from the physical device to the emulator. In consequence, also the interrupt controller needs to be instantiated in the emulated platform to correctly relay the interrupt signal from the timer to the processor. Now, our system has one interrupt controller on the physical platform, and one in the emulator. Whenever an interrupt occurs on in the emulator, it is obvious that this interrupt stems from the migrated timer, and that the emulated interrupt controller needs to be accessed to execute the correct interrupt handler. However, if an interrupt occurs on the physical platform, say, due to a serial port receiving data, then the interrupt controller on the

physical platform needs to be interrogated to find the right interrupt handler. Hence, our system needs to keep track of where an interrupt was generated, and have enough knowledge about the interrupt controller to forward accesses to the right instance.

The same considerations also apply for other peripherals with direct connections to other peripherals, like the DMA controller and the General-Purpose Input Output (GPIO) controller. For the DMA controller, the added challenge consists in understanding if a physical memory range has been modified by the emulated DMA controller in the emulator, or a memory range on the physical platform has been updated by the physical DMA controller.

### 5.5.2 Conclusion

Whole-platform emulation is a challenging topic, and has been driven mostly by manual emulator coding. In this chapter, we proposed a methodology to automatically infer a platform's peripheral devices, and use this information to generate a platform description. With some added manual work, this platform description can then be used to automatically generate an emulator. We described the various challenges we met during the implementation of this work: sufficient exercising of peripherals by different firmware, making a large number of platforms work in our instrumented environment, choosing features to measure, and finally mining information from gathered features to identify peripherals. While there is still room for improvement in the current fingerprint comparison heuristics, we believe that the methodology is valid and worth pursuing further. Adding more information, e.g., from static code analysis, could be used to enhance the peripheral fingerprints further.

The code written during this project will be contributed back to the community once this work has been released.

# Chapter 6

# Conclusion

In this final chapter, we first want to discuss future work to extend *Avatar* and the peripheral identification system.

### 6.0.3 Future Work

Future work on *Avatar* will consist in integrating better analysis techniques to achieve better automated bug detection. Especially data structure recovery, as presented in Howard [SSB11], could help to find memory-related bugs earlier and more reliably. Combined with ideas from Dowser [HSNB13], a white-box fuzzer to find buffer boundary violations, *Avatar* could go deeper in code exploration than it is currently capable of. Exploit generation techniques as used in AEG [ACHB11] and access control methods as presented in Firmalice [SWH+15] could be applicable as well.

Our peripheral identification system could be extended to not only gather static fingerprints, but also stateful information about a peripheral. Currently, some devices will be modelled incorrecty, as the meaning of registers changes depending on the configuration. Our fingerprint is currently not capable of expressing this information. A peripheral description in form of a state machine would be even more beneficial if a device emulation could be generated right from the description, without the need of a hand-coded emulated device. We imagine that it is feasible to extract a model of the expected peripheral device's behaviour from firmware code. Device drivers could be identified with static analysis, for example, and then exercised with symbolic execution to understand expected device behaviors. Such a model might not be 100% exact, but at least useful enough generate a fake peripheral which can lead to an exploration of deeper firmware code regions. Especially for special-purpose peripherals, such as the servo controller in the analyzed hard drive, this might be the only way to generate a complete emulator.

## 6.1 Conclusion

This dissertation presented novel methods for the dynamic analysis of binary firmware. We designed *Avatar*, a tool for the execution of firmware code in an analysis-friendly emulator. The system orchestrates the emulator and the physical device to avoid having to emulate all the platform's peripherals. Instead, accesses to peripherals are forwarded to the physical device. In this way, hardware platforms with unknown peripherals can be analyzed.

Then, we applied *Avatar* to the analysis of a hard disk drive firmware. With knowledge gathered through *Avatar*'s assisted reverse engineering, we could inject a backdoor into the firmware which intercepts and modifies read and write operations on the disk. Our backdoor can be installed in little time and has a very small overhead that will go unnoticed in day-to-day operations. Parallel work of other researchers and revelations about spying tools employed by the NSA have confirmed our point that such a backdoor can be implemented by a moderately funded researcher, and a fortiori by a well-funded nation state.

Finally, we designed a peripheral identification system working on top of *Avatar*. By moving peripherals from the physical platform to the emulator, we remove some of the limitations *Avatar* is exhibiting in whole-system analysis. We demonstrated the system's concept on eight different platforms, and showed that an emulator can be reconstructed from the gathered information.

# Appendix A

# Résumé en français

Les systèmes embarqués sont devenus de plus en plus omniprésents dans nos vies. Des systèmes industriels ne peuvent guère être conçu sans contrôle numérique, des voitures contiennent des dizaines d'unités de contrôle électroniques et des millions de lignes de code, et la domotique se répand de plus en plus.

Bien que la corruption un compromis d'un ordinateur personnel ou d'un serveur puisse entraîner des graves problèmes et des pertes importantes, des systèmes embarqués piratés peuvent avoir des conséquences encore plus sévères.

Les systèmes embarqués sont typiquement utilisés pour surveiller et contrôler des processus du monde physique, où ils peuvent endommager des hommes et de l'équipement. Le ver *Stuxnet*, qui a infecté des postes de travail dans une usine d'enrichissement d'uranium en Iran en 2010, est un bon exemple de ceci. En injectant des commandes malveillantes dans le programme de contrôle des centrifuges, il est fortement probable que le ver a détruit plusieurs centaines de centrifuges et a nettement retardé le programme nucléaire iranien [NFC11, Lan13].

Jusqu'à récemment, les systèmes embarqués avaient pour la plupart été découplés de l'Internet. Cependant, les appareils embarqués connectés appelés «appareils intelligents», qui ne traitent pas seulement les entrées sensorielles locales, mais reçoivent aussi des données provenant d'autres systèmes, deviennent de plus en plus commun. Le réseau électrique ne pouvait pas gérer les grandes fluctuations en production d'électricité, qui arrivent avec la production solaire, sans avoir des prévisions météorologiques. Les processus industriels sont censés de gagner en efficacité avec ce qui est appelé «l'Industrie 4.0» ou «l'Internet des objets industriels connectés (IIoT)». Et même dans nos foyers, le chauffage, la lumière et les serrures seront bientôt connectés dans «l'internet des objets (IoT)».

Le nombre réel d'attaques contre les systèmes embarqués est difficile à estimer. Rapports des intrusions spectaculaires dans les médias, comme Stuxnet

ou l'effondrement d'une fournaise dans une aciérie allemande malicieusement induite [bsi15], soulignent des évènements isolés. Or, les exploits des systèmes embarqués, comme des terminaux de cartes de crédit [NB14], des contrôleurs industriels programmables (PLC) utilisé pour le contrôle des processus de production dans les usines [Ber11], et prises de courant disjonctable à distance [Dav14], sont souvent découverts et présentés lors de conférences de sécurité. La base de données NIST des vulnérabilités ne contenait que 190 entrées concernant les logiciels embarqués en 2010, tandis que le nombre a multiplié par dix à environ 1700 en 2014.

Tous ces points mettent en avant que des appareils embarqués connectés peuvent causer des dommages physiques quand ils ne sont pas sécurisé convenablement. En conséquence, les gouvernements sont de plus en plus sensibles à la sécurité des systèmes embarqués. surtout quand ils sont utilisés dans des «infrastructures critiques», comme l'électricité, et la distribution du carburant et de l'eau [nis14, bsi15, ftc15].

## Un manque de sécurité dans les systèmes embarqués

Les systèmes embarqués ne montrent actuellement pas la même résistance contre des attaques comme, par exemple, les systèmes PC le font. Pour cela, il y a plusieurs raisons.

Premièrement, des systèmes embarqués sont généralement utilisé sur une longue période de temps. Au cours de leur durée de vie, les techniques d'attaques et les menaces évoluent de manière significative, tandis que le logiciel de l'appareil, appelé firmware, est rarement ou jamais mis à jour. Par exemple, un grand nombre d'appareils grand public connecté à l'Internet utilisent un noyaux Linux obsolète avec des vulnérabilités connues [CZFB14].

Deuxièmement, l'interconnectivité est parfois ajoutée à partir d'un concept de sécurité préexistant, ce qui entraîne des problèmes de sécurité imprévues. Bien que l'extension d'interfaces existantes est une manière facile de connecter du matériel plus âgé dans le même système, l'exemple de Modbus illustre les dangers de cette approche. Modbus [spe12] est un bus industriel qui a été conçu pour connecter de l'équipement local dans une usine. En 1999, un transport du protocole ModBus était ajouté au bus, sans ajouter des dispositifs pour la sécurité et l'authentification. En conséquence, l'analyse de l'espace routable d'adressage IPv4 a révélé plus de 12,000 périphériques ModBus qui sont directement connectés à l'Internet et accessible à tous [Lal15].

Troisièmement, les fabricants des systèmes embarqués ont reçu peu de pression de leurs clients pour fournir des systèmes sécurisés, et ont mis l'accent de développement sur des nouvelles fonctionnalités et la sécurité des utilisateurs. Les appareils grand public tels qu'un téléviseur «smart», où une nouvelle fonc-

tionnalité est plus probable à influencer la décision d'achat qu'un certificat de sécurité, en sont un exemple.

Quatrièmement, le développement de dispositifs embarqués est entraîné par des considérations de coût et le «time-to-market». Des fonctions de sécurité, qui auront besoin de plus de surface de silicium pour leur fonctionnement, entraînent des coûts augmentés. Des mesures de sécurité ajouté au logiciel nécessitent normalement plus de puissance CPU et de mémoire, ce qui se traduit par une puce plus chère. Des systèmes qui fonctionnent sur une alimentation limitée, comme des nœuds d'un réseau de capteurs alimentés par batterie, donc sacrifier la sécurité pour moins de consommation électrique.

Enfin, une conception robuste de sécurité nécessite un effort de développement supplémentaire. Surtout pour les produits de vente de masse et les produits grand public, où un prix bas est un facteur décisif dans la décision d'achat, les fabricants sont tentés de renoncer à la sécurité.

**Des systèmes embarqués sont des cibles d'attaque attractifs** Les systèmes embarqués constituent une cible attractive pour les attaquants. En raison d'un manque de gestion de logiciel embarqué, ils reçoivent rarement des mises à jour aussi fréquents que les systèmes PC [CZFB14]. En plus, surtout le matériel plus âgé est livré avec une configuration peu sécurisée par défaut (p.ex., des mots de passe qui sont facile à deviner), qui est amélioré seulement par peu d'utilisateurs après. Encore pire, une fois qu'on matériel embarqué est infecté avec un logiciel malveillant, il est aujourd'hui très difficile de détecter cette infection – tant que la fonctionnalité du matériel ne change pas visiblement, très peu d'utilisateurs vont suspecter une activité malveillante. Et même si un logiciel malveillant est trouvé, il est presque impossible de l'enlever d'une manière fiable.

Le fait que le logiciel embarqué soit peu mis à jour et fortifié contre des attaques, et que les systèmes embarqués sont, contrairement aux PCs, toujours allumés, les rend une cible attractive pour les cyber-criminels. Bien que l'hétérogénéité des plates-formes pose un défis aux auteurs des logiciels malveillants, des kits d'exploitation facilitent le développement du code malveillant en donnant des outils d'identification des systèmes et une couche d'abstraction. Une vague de vers pour les routers résidentiels illustrent ce point: soit le logiciel embarqué ou la configuration sont changés pour injecter des publicités dans les sites web [Fra15], voler des données d'accès bancaires [cym14], ou faire parti d'un botnet [GAZ15, Car13].

Les systèmes embarqués industriels, qui sont généralement séparés de l'Internet par la ségrégation de réseau, sont néanmoins confrontés aux mêmes défis de la gestion du logiciel et la détection des logiciels malveillants. Les scénarios d'attaques les plus probables pour les systèmes industriels sont des APTs, comme la pénétration d'un réseau d'entreprise nécessite des compétences et des efforts plus dévoués. Dans ces attaques, des attaquants de premier plan (par exemple,

des états ou des organisations terroristes) concentrer leurs efforts malveillants contre un système particulier (par exemple, une centrale électrique) [NFC11, Lan13].

**Des tests et de l'assistance à la retro-conception sont nécessaires pour les systèmes embarqués** Les points précédents soulignent la nécessité d'avoir des outils et techniques pour améliorer la sécurité des systèmes embarqués. Même si des parties ou l'ensemble du code source d'un logiciel embarqué est disponible, l'analyse du code binaire peut être la seule option pour un tiers, comme la chaîne des outils pour la compilation du logiciel embarqué normalement n'est pas fourni. De plus, quand la compilation du code source ne mène pas à exactement la même représentation binaire, ce qui peut arriver à cause d'une autre version de compilateur ou une configuration différente, il est extrêmement difficile de prouver que le code source correspond au logiciel binaire original. Ainsi, il peut être plus facile d'analyser le firmware binaire en premier lieu. En général, l'analyse binaire du code binaire embarqué joue un rôle très important dans plusieurs scénarios.

Tout d'abord, après un incident de sécurité, une analyse post-mortem est nécessaire pour le comprendre et d'améliorer les pratiques de sécurité ensuite. Ce travail est généralement effectuée par des enquêteurs qui ne disposent pas d'accès au code source. Toute assistance automatisé à l'analyse pourrait aider à accélérer ce processus.

Après, les fabriquants ou leurs clients peuvent nécessiter des tests de pénétration. Deuxièmement, les fabricants et les clients ont parfois besoin d'effectuer des tests de pénétration. Dans ce cas, le matériel est typiquement testé sans connaissances de leur fonctionnement intérieur («black-box»). Ses interfaces sont testé avec des valeurs invalides et des cas limites pour les protocoles connus, mais aucune connaissance du logiciel est utilisé dans ce processus. En profitant des approches telles que le *fuzzing guidé* où les connaissances du logiciel embarqué sont utilisés, des tests de pénétration pourraient être rendus plus efficace.

Enfin, l'évaluation suffisante des appareils embarqués est particulièrement importante pour les fabricants intégrateurs. Lorsque plusieurs appareils embarqués sont intégrés dans un système, les tests avec connaissances du fonctionnement intérieur («white-box») sont encore plus important. Chaque faille individuelle d'un matériel peut mener à une situation de compétition ou un blocage qui arrive très rarement et seulement avec des circonstances particulières.

**Les outils d'analyse statique et dynamique nécessitent d'être adaptés** L'analyse binaire pour le logiciel PC a considérablement progressé dans les dernières dix années. Le fuzzing «white-box» des applications et des interfaces du système d'exploitation, la vérification des traces pour traquer les flux

des données à travers de l'application, et l'exécution symbolique pour découvrir automatiquement des tests et augmenter la couverture du code sont des techniques avancées d'analyse dynamique qui sont utilisées fréquemment par des experts. Malheureusement, ces techniques ne peuvent être appliquées à des systèmes embarqués.

Une des raisons pour ceci est l'hétérogénéité de logiciel de système dans les systèmes embarqués. Alors que certains systèmes ont une séparation claire et une interface bien définie entre le système d'exploitation et l'application, comme les systèmes basés sur Linux, d'autres utilisent des systèmes d'exploitation propriétaires. Surtout avec les systèmes de gamme moins chère et professionnelle, on trouve un logiciel monolithique où l'application est compilé ensemble avec une bibliothèque de système d'exploitation.

En outre, en fonction du système embarqué, les caractéristiques de sécurité du matériel changent beaucoup. Processeurs ARM de haute gamme ont plusieurs niveaux de privilèges et support pour la virtualisation du matériel, tandis que les processeurs de basse gamme de la même famille ne peuvent distinguer que trois niveaux de privilèges et ne supportent pas la notion de mémoire virtuelle.

L'analyse statique est rendue difficile par les nombreux jeux d'instructions employés dans les processeurs des systèmes embarqués. ARM est le jeu d'instructions le plus répandu [CZFB14], mais MIPS, AVR, MSP430, et 8051 sont aussi fréquemment utilisés. Le développement d'outils pour un seul jeu d'instructions n'est pas une solution viable si veut analyser le logiciel embarqué à grande échelle.

Finalement, des méthodes d'analyse dynamique qui vont plus loin que le débogage et le traçage sont difficile à utiliser avec le logiciel embarqué. Sans matériel spécialisé, des techniques plus avancées nécessitent un émulateur instrumenté qui exécute le firmware. Cependant, le fonctionnement du logiciel et du matériel sont liés en sorte qu'un logiciel embarqué ne fonctionnera pas correctement si le matériel périphérique du système n'est pas émulé. Ainsi, il est nécessaire de faire une retro-conception du système à la base d'une analyse statique pour être capable de construire un émulateur et de conduire une analyse dynamique avancée.

## A.1 Énoncé du problème

Cette thèse est centrée autour du problème de l'analyse du dynamique du logiciel embarqué binaire. Des analystes indépendants tels que les laboratoires de certification, les testeurs de pénétration, analystes judiciaires, les fabricants qui composent un système plus complexe à partir de plusieurs plus petits appareils, et les chercheurs ont un intérêt légitime d'analyser du logiciel embarqué pour des raisons de sécurité. Mais l'analyse du firmware binaire peut s'avérer difficile: Émulateurs doivent spécifiquement être adapté à chaque matériel périphérique,

comme aucune abstraction commune du matériel n'existe. Même quand on a un moyen de déboguer un logiciel embarqué sur son matériel, l'application des méthodes d'analyse dynamique moderne comme exécution symbolique est impossible, car le soutien du matériel serait nécessaire. Au lieu de cela, une grande partie de temps d'analyse est investi à retracer les plantages et de suivre le flux des donnés à la main. Même le système d'exploitation, l'application et des bibliothèques dans un ensemble du code binaire doivent être identifiés de nouveau à chaque fois. Avec un environnement qui permet analyse des flux de données, le traçage de l'exécution et de l'instrumentation, plus de ces tâches pourraient être automatisées.

## A.2 Contributions

Nous proposons un ensemble d'outils qui comble ce manque d'outils d'analyse dynamique pour les systèmes embarqués. *Avatar*, tel que présenté dans le Chapitre 3, permet d'émuler le logiciel d'un système embarqué. Des analyses complexes tels que l'exécution concolique peut être implémenté sur cette fondation. Nous présentons plusieurs techniques qui peuvent être utilisées pour optimiser la performance du système, et d'adapter *Avatar* aux besoins de l'utilisateur. *Avatar* est démontré dans trois différents scénarios d'analyse de sécurité, y compris la retro-conception, la découverte des vulnérabilités, et la détection des portes dérobées. Pour démontrer la flexibilité de notre système, chaque test a été effectué sur une catégorie de système embarqué différente.

En outre, dans le chapitre 4, nous présentons une porte dérobée implanté dans le logiciel embarqué d'un disque dur SATA grand public. À partir de cet exemple, nous démontrons les dangers et la perte catastrophique de sécurité dû aux modifications malveillantes du logiciel embarqué. La porte dérobée est autonome, ne nécessitant aucune coopération de l'hôte. Elle est bien caché. de part le fait qu'elle n'intercepte que les lectures et écritures légitimes, sans se baser sur le DMA et d'autres caractéristiques avancées. Le temps additionnel requis pour le fonctionnement de la porte dérobée est imperceptible par l'utilisateur pendant l'utilisation normale. De plus, la porte dérobée peut être installé par un logiciel sur l'hôte dans très peu de temps. Nous démontrons également qu'il est possible de construire une telle porte dérobée avec un investissement d'une dizaine de mois-homme, malgré les difficultés de débogage et de la retro-conception du logiciel embarqué du disque. Enfin, nous présenter un certain nombre de techniques d'analyse judiciaire qui peuvent aider à identifier une telle porte dérobée.

Finalement, dans chapitre 5 nous proposons une méthodologie pour identifier la configuration du matériel périphérique d'un système embarqué. *Avatar* aborde le problème du couplage étroit entre le logiciel et le matériel en relayant les accès au matériel de l'émulateur, où le logiciel tourne, au plate-forme physique. Cette approche, qui est bien adaptée pour l'analyse des petits bouts de code, entraîne

quelques désavantages: La simulation du système entier est difficile, comme du code avec des contraintes de temps réel doit être identifié, des interruptions doivent être gérées correctement, etc. En plus, l'écriture des valeurs concrètes au matériel physique pendant l'exécution symbolique rend tous les autres états symboliques invalides. Dans cet ouvrage, nous observons la communication entre le logiciel et les périphériques dans $S^2E$ pour créer une empreinte digitale de chaque périphérique, similaire aux descriptions des registres dans les fiches produit. Nous démontrons qu'une base d'empreintes peut être construite, à partir de laquelle une recommandation automatique pour un plan des périphériques peut être donné.

## A.3 Organisation de la thèse

Dans cette thèse, nous analysons la faisabilité et l'impact d'une attaque ciblée contre un système embarqué, plus spécifiquement un disque dur, et de développer des outils d'analyse dynamique binaire pour analyser de telles menaces. Après avoir résumé l'état de l'art dans Chapitre 2, nous présentons *Avatar*, un outil d'analyse dynamique pour les systèmes embarqués, dans le chapitre 3. Dans le chapitre suivant, la conception et la mise en œuvre d'une attaque de modification de logiciel embarqué est démontré. Chapitre 5 présente ensuite des techniques de retro-conception automatisé basées sur *Avatar*. Enfin, la thèse conclut dans chapitre 6.

**Chapitre 2 – Literature review** Chapitre 2 résume l'état des techniques le plus récentes. Un aperçu de l'analyse binaire statique, dynamique et symbolique est donnée, qui est pertinent pour tous les chapitres suivants. La section sur la sécurité des logiciels embarqués est surtout pertinente pour les chapitres 3 et 4. La résumé sur les portes dérobées est liée aux travaux présentés dans chapitre 4, et enfin la retro-conception des pilotes et du matériel est lié á chapitre 5.

**Chapter 3 – *Avatar*: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares** Dans ce chapitre, nous présentons Avatar, un ensemble d'outils d'analyse dynamique binaire pour les systèmes embarqués. L'introduction donne un aperçu de la conception des systèmes embarqués et des défis dans l'émulation du logiciel embarqué. À la suite, l'idée essentielle d' *Avatar*, le renvoi des accès I/O de l'émulateur au matériel physique, est expliqué. Plusieurs techniques pour améliorer la performance du système sont discutées, et sont démontrés sur trois cas d'utilisation différents. La publication sur laquelle ce chapitre se base a été publié à la conférence NDSS en 2014 [ZBFB14].

**Chapter 4 – Implementation and Implications of a Stealth Hard-Drive Backdoor** Ici, nous présentons une porte dérobée d'exfiltration des données implantée dans le logiciel embarqué d'un disque dur. L'introduction motive pourquoi un disque dur est une cible d'attaque attractive, et établit le modèle de menace. Ensuite, la retro-conception du logiciel embarqué du disque et l'implémentation de la porte dérobée sont discutés. L'exfiltration des données sans assistance du système d'exploitation de l'hôte est exposé, et la performance de la porte dérobée est évaluée. Le travail de ce chapitre a été présenté à la conférence ACSAC en 2013 [ZKB+13].

**Chapter 5 – Towards automating platform reverse engineering of embedded devices** Ce chapitre propose une technique pour la retro-conception de la plate-forme matérielle d'un système embarqué. Un aperçu des défis de l'analyse d'un système entier avec *Avatar* est donné. Puis, une technique pour collecter des empreintes digitales des accès du logiciel embarqué au matériel est décrite. Enfin, les limitations et les futurs améliorations de cette méthode sont discutées.

**Chapter 6 – Future work and conclusion** Enfin, nous présentons les problèmes de recherche, et nous concluons la thèse avec un bilan des contributions des chapitres précédents.

# Glossary

**8051** Instruction set used by 8-bit Intel microncontrollers of the MCS-51 family. 4, 93

**ACPI** Advanced Configuration and Power Interface. 70

**AHB** AMBA High-performance Bus. 70

**AMBA** Advanced Microcontroller Bus Achitecture. 70

**APB** Advanced Peripheral Bus. 70

**APT** Advanced Persistent Threat. 3, 91

**ARM** An instruction set architecture designed by ARM (formerly Acorn RISC Machines). Instruction words are 4 bytes long, but a compressed version called *Thumb* exists. ARM does not produce CPUs themselves, but licenses IP to other manufacturers. 4, 73, 93

**ASIC** Application-Specific Integrated Circuit. 19, 70

**AVR** Instruction set used by smaller 16-bit Atmel microcontrollers. 4, 93

**AXI** Advanced eXtensible Interface. 70

**BAP** Binary Analysis Platform. 10

**CPU** Central Processing Unit. 2, 19, 91, 99

**DBI** Dynamic Binary Instrumentation. 12

**DMA** Direct Memory Access. 70, 73, 74, 86

**DSP** Digital Signal Processor. 19

**DTB** Device Tree Blob. 70

**DWT** Data Watchpoint and Trace. 78

**ESIL** Evaluable Strings Intermediate Language. 10

**ETM** Embedded Trace Macrocell. 78

**FDT** Flattened Device Tree. 70

**firmware** The set of all executable code (both operating system and applications) running on an embedded device. 2, 90

**FPGA** Field Programmable Gate Array. 77, 78

**GPIO** General-Purpose Input Output. 86

**GSM** Global System for Mobile Communications. 20

**HDD** Hard Disk Drive. 6, 69

**I/O** Input-Output. 7, 69, 72, 73, 95

**IIoT** Industrial Internet of Things. 1

**IoT** Internet of Things. 1

**IP** Intellectual Property. 70

**IPv4** Internet Protocol Version 4. 2, 90

**IR** Intermediate Representation. 10

**LLVM** Originally called Low-Level Virtual Machine, a framework that is now a compiler construction toolbox. Any programming language is translated to an intermediate language by a frontend, before being lowered to machine code. 10

**MBR** Master Boot Record. 49

**MIPS** MIPS (from Microprocessor without Interlocked Pipeline Stages) is an instruction set architecture owned by Imagination Technologies. 4, 93

**MMIO** Memory-Mapped Input-Output. 72–74, 77

**MSP430** Instruction set used by smaller 16-bit Texas Instruments microcontrollers. 4, 93

**NIST** National Institute of Standards and Technology. 2, 90

**PC** Personal Computer. 3, 4, 91, 92

**PCI** Peripheral Component Interconnection. 70, 74

**PLC** Programmable Logic Controller. 1

**REIL** Reverse Engineering Intermediate Language. 10

**RREIL** Relational Reverse Engineering Intermediate Language. 10

**SBI** Static Binary Instrumentation. 12, 13

**SDK** Software Development Kit. 70

**SoC** System on Chip. 17, 19, 20, 22, 70

**SSA** Static Single Assignment. 11

**TCG** Tiny Code Generator, Qemu's internal intermediate representation of translated machine code. 11

**TCP** Transmission Control Protocol. 2

**TI** Texas Instruments. 70, 79

**USB** Universal Serial Bus. 20

**VM** Virtual Machine. 10

**x86** Intel x86 CPU architecture. 73

# List of Figures

# List of Tables

# List of Publications 2011—2015

## Conference and Journal Publications

1. J. Zaddach, D. Balzarotti, A. Francillon, *"Towards automating platform reverse engineering of embedded devices"*, To be submitted to Design Automation and Test Conference, March 2016, Dresden, Germany.

2. L. Cojocar, J. Zaddach, H. Bos, D. Balzarotti, A. Francillon, *"PIE: Parser Identification in Embedded Systems"*, Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), December 2015, Los Angeles, CA, USA.

3. A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, *"A Large Scale Analysis of the Security of Embedded Firmwares"*, Proceedings of the 23rd USENIX Conference on Security Symposium, August 2014, San Diego, CA, USA.

4. J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, *"Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares"*, Network and Distributed System Security (NDSS) Symposium , February 2014, San Diego, CA, USA.

5. J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, I. Koltsidas, *"Implementation and Implications of a Stealth Hard-Drive Backdoor"*, Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC), December 2013, New Orleans, LA, USA.

6. M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, S. Loureiro, *"A security analysis of amazon's elastic compute cloud service"*, Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC), March 2012, Trento, Italy.

## Invited Presentations, Poster Presentations, and Short Papers

1. J. Zaddach, *"Avatar: A Framework to Support Dynamic Security Analysis of Embedded System's Firmwares"*, Presentation at SSTIC, June 2015, Rennes, France.

2. A. Costin, J. Zaddach, *"firmware.re - firmware unpacking, analysis and vulnerability-discovery as a service"*, Invited presentation at SECURE, October 2014, Warshaw, Poland.

3. J. Zaddach, *"firmware.re - firmware unpacking, analysis and vulnerability-discovery as a service"*, Presentation at Black Hat Europe, October 2014, Amsterdam, The Netherlands.

4. J. Zaddach, L. Bruno, *"AVATAR: A Framework for Dynamic Security Analysis of Embedded Systems' Firmwares"*, Invited presentation at École Polytechnique de Lausanne, January 2014, Lausanne, Switzerland.

5. J. Zaddach, *"Exploring the impact of a hard drive backdoor"*, Presentation at Recon, June 2014, Montréal, Canada.

6. J. Zaddach, *"An Amazing Journey into the depth of my Hard Drive"*, Presentation at Power of Community, November 2013, Seoul, South Korea.

7. A. Costin, J. Zaddach, *"Embedded Devices Security Firmware Reverse Engineering"*, Workshop presentation at Black Hat, August 2013, Las Vegas, NV, USA.

# Bibliography


[ACHB11] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Network and Distributed System Security Symposium*, Feb. 2011, pp. 283–300. 36, 87

[AGG+15] J. Appelbaum, A. Gibson, C. Guarnieri, A. Müller-Maguhn, L. Poitras, M. Rosenbach, L. Ryge, H. Schmundt, and M. Sontheimer, "The Digital Arms Race: NSA Preps America for Future Battle," Jan. 2015. [Online]. Available: http://www.spiegel.de/international/world/new-snowden-docs-indicate-scope-of-nsa-preparations-for-cyber-battle-a-1013409.html last accessed on 2015-06-14. 49

[AiC] S. Àlvarez i Capilla. radare2 webpage. [Online]. Available: http://www.radare.org/r/ last accessed on 2015-05-31. 10

[anu] Anubis – Malware Analysis for Unknown Binaries. [Online]. Available: http://anubis.iseclab.org/ last accessed on 2015-05-31. 20

[ARM04] ARM, "ARM966E-S, Revision: r2p1, Technical Reference Manual," 2004. [Online]. Available: http://infocenter.arm.com 55

[aS] J. D. (a.k.a. Sprite_tm). HDD Hacking. Talk given at OHM 2013. [Online]. Available: http://spritesmods.com/?art=hddhack last accessed on 2015-05-31. 16, 50

[ASKE10] K. Anand, M. Smithson, A. Kotha, and K. Elwazeer, "Decompilation to compiler high ir in a binary rewriter," University of Maryland, Tech. Rep., Nov. 2010. [Online]. Available: http://www.ece.umd.edu/~barua/high-IR-technical-report10.pdf last accessed on 2015-05-31. 11

[bac12] backupworks.com, "HDD Market Share - Rankings in 2Q12," 2012. [Online]. Available: http://www.backupworks.com/hdd-market-share-western-digital-seagate.aspx 50

[BBB09] H. Bojinov, E. Bursztein, and D. Boneh, "Embedded management interfaces: Emerging massive insecurity," in *Blackhat 2009 Technical Briefing / whitepaper*, 2009. 20

[BCdJ+06] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, ser. VEE '06. New York, NY, USA: ACM, 2006, pp. 154–163. [Online]. Available: http://doi.acm.org/10.1145/1134760.1220164 12

[BDA01] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dynamic optimization framework for windows," in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001. 13

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/800027.808445 14

[Bel05] F. Bellard, "QEMU, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. 13, 14, 24

[Ber11] D. Beresford, "Exploiting siemens simatic s7 plcs," NSS Labs, Tech. Rep., Jul. 2011. 1, 90

[BFS04] F. Baker, B. Foster, and C. Sharp, "Cisco Architecture for Lawful Intercept in IP Networks," RFC 3924 (Informational), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3924.txt 50

[BJAS11] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 463–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_37 10

[BM11] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*,

ser. PASTE '11. New York, NY, USA: ACM, 2011, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/2024569.2024572 13

[BMMS11] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 12–22. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001423 14

[BR12] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-Bound Encryption," in *Annual Computer Security Applications Conference*, Orlando, USA, 2012, pp. 71–78, ISBN 978-1-4503-1312-4. 65

[bsi15] "Die lage der it-sicherheit in deutschland 2014," Bundesamt für Sicherheit in der Informationstechnik, Tech. Rep., 2015. 1, 2, 90

[BYXO+07] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn, "The Price of Safety: Evaluating IOMMU Performance," in *The Ottawa Linux Symposium*, Ottawa, Canada, 2007, pp. 9–20. 65

[Car13] Carna Botnet, "Internet Census 2012 — Port Scanning /0 Using Insecure Embedded Devices," Tech. Rep., Mar. 2013. [Online]. Available: http://census2012.sourceforge.net/paper.html 3, 20, 91

[CARB12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.31 14

[CC10a] V. Chipounov and G. Candea, "Dynamically Translating x86 to LLVM using QEMU," École Polytechnique Fédérale de Lausanne, Tech. Rep., Mar. 2010. [Online]. Available: http://infoscience.epfl.ch/record/149975/files/x86-llvm-translator-chipounov_2.pdf last accessed on 2015-05-31. 11

[CC10b] ——, "Reverse Engineering of Binary Device Drivers with RevNIC," in *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), Paris France, April 2010*, Paris, France, 2010. 17, 22, 35

[CC11] ——, "Enabling sophisticated analyses of x86 binaries with revgen," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, Jun. 2011, pp. 211–216. 11

[CCK94] P. C. Ching, Y. Cheng, and M. H. Ko, "An in-circuit emulator for TMS320C25," *IEEE Transactions on Education*, vol. 37, no. 1, pp. 51–56, 1994. 21

[CCS13] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation." in *NDSS*. The Internet Society, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#CuiCS13 16, 20

[CDE08a] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756 13, 14

[CDE08b] ——, "KLEE unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008. 20, 24

[CF10] A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 585–594. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866373 14

[CFH+05] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251223 27

[CFPS09] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Conference on Computer and Communications Security*, Chicago, USA, 2009, pp. 400–409, ISBN 978-1-60558-894-0. 67

[CFRN95] R. A. Caldeira, J. C. Fravel, R. G. Ramsdell, and R. N. Nolasco, "Hard disk drive architecture," Patent US 5 396 384, Jul. 03, 1995.

[Online]. Available: http://www.freepatentsonline.com/5396384.html 52

[CKC12] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E Platform: Design, Implementation, and Applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110358 14, 20, 24, 33

[CMA+11] S. Checkoway, D. McCoy, D. Anderson, B. Kantor, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analysis of Automototive Attack Surfaces," in *Proceedings of the USENIX Security Symposium*, San Francisco, CA, Aug. 2011. 20

[Cop14] P. T. Copeland, "Using state merging and state pruning to address the path explosion problem faced by symbolic execution," Ph.D. dissertation, DTIC, Jun. 2014. 13

[CPC+08] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 391–402. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455820 18

[Cri82] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, vol. C-31, no. 6, pp. 531–540, 1982. 35

[Cro10] T. Cross, "Exploiting Lawful Intercept to Wiretap the Internet," Black Hat, Las Vegas, USA, 2010. [Online]. Available: http://www.blackhat.com/ 50

[CS98] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, 1998, pp. 126–133. [Online]. Available: http://dx.doi.org/10.1109/WPC.1998.693332 10, 14

[CS11] A. Cui and S. J. Stolfo, "Defending embedded systems with software symbiotes," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 358–377. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_19 13, 15, 22

[CSXK08] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 255–266. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855759 36

[CWKK09] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 2009, pp. 110–125. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.14 18

[CWS08] (2008) CWSandbox. Http://www.cwsandbox.org. 20

[CYLS07] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 317–329. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315286 17

[cym14] "SOHO Pharming – A Team Cymru EIS Report: Growing Exploitation of Small Office Routers Creating Serious Risks," Team Cymru, Tech. Rep., Feb. 2014. [Online]. Available: https://www.team-cymru.com/ReadingRoom/Whitepapers/2013/TeamCymruSOHOPharming.pdf 3, 91

[CZFB14] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 95–110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin 2, 3, 4, 90, 91, 93

[Dae] Daemon9, "Project Loki," Phrack 49. [Online]. Available: http://www.phrack.org/issues.html?id=6&issue=49 16

[Dav14] M. Davis, "Belkin WeMo Home Automation Vulnerabilities," IOActive, Tech. Rep., Feb. 2014. 1, 90

[Del] G. Delugré, "Closer to metal: Reverse engineering the broadcom netextreme's firmware," HACK.LU 2010. 16, 20

[DGHH+14] B. F. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering for the greater good with panda," Columbia University, Tech. Rep., 2014, part number CUCS-022-14. [Online]. Available: http://dx.doi.org/10.7916/D8WM1C1P 12

[DLHL13] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 839–850. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516697 12

[DMJR13] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the USENIX Security Symposium*, Washington, DC, Aug. 2013. 15, 20, 22

[Dor04] M. Dornseif, "Owned by an iPod: Firewire/1394 Issues," 2004. [Online]. Available: http://md.hudora.de/presentations/firewire/PacSec2004.pdf 65

[DR14] A. Dinaburg and A. Ruef, "McSema: Static Translation of X86 Instructions to LLVM," Presentation slides, Jun. 2014, presented at Recon, Montreal, Canada. [Online]. Available: https://www.trailofbits.com/resources/McSema.pdf 11

[ESKK08] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126 20

[FBG12] P. Feiner, A. D. Brown, and A. Goel, "Comprehensive kernel instrumentation via dynamic binary translation," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150992 13

[FMC11] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier," 2011. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf 20

[Fra15] S. Frankoff, "Ad-fraud malware hijacks router dns - injects ads via google analytics," Blog post, Mar.

2015. [Online]. Available: http://sentrant.com/2015/03/25/ad-fraud-malware-hijacks-router-dns-injects-ads-via-google-analytics/ last accessed on 2015-05-27. 3, 91

[Fre11a] Freescale Semiconductor, Inc., "MC1322x Simple Media Access Controller Demonstration Applications User's Guide," 9 2011, rev. 1.3. 43

[Fre11b] ——, "MC1322x Simple Media Access Controller (SMAC) Reference Manual," 09 2011, rev. 1.7. 43

[ftc15] "Internet of things: Privacy and security in a connected world," Federal Trade Commission, Tech. Rep., Jan. 2015. 2, 90

[GAZ15] O. Gayer, R. Atlas, and I. Zeifman, "Lax security opens the door for mass-scale abuse of soho routers," Blog post, May 2015. [Online]. Available: https://www.incapsula.com/blog/ddos-botnet-soho-router.html last accessed on 2015-05=27. 3, 91

[GH06] D. Gibson and B. Herrenschmidt, "Device trees everywhere," OzLabs, IBM Linux Technology Center, Tech. Rep., Feb. 2006. 70

[GKS05] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065036 14

[GLM08] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008. [Online]. Available: http://www.truststc.org/pubs/499.html 14, 20

[GLM12] ——, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081 14, 20

[HA10] A. Heydari and S. Azimi, "A survey in deterministic replaying approaches in multiprocessors," *International Journal of Electrical & Computer Sciences IJECS-IJENS*, vol. 10, no. 4, 2010. 12

[HA14] C. Heitman and I. Arce, "Barf: A multiplatform open source binary analysis and reverse engineering framework," in *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*, 2014. 10

[hdd13] "HDD Guru Forums," 2013. [Online]. Available: http://forum.hddguru.com/ 54

[HDK+11] O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, USA, 2011, pp. 279–290, ISBN 978-1-4503-0266-1. 51

[Hea06] J. Heasman, "Implementing and Detecting an ACPI BIOS Rootkit," Black Hat, Las Vegas, USA, 2006. [Online]. Available: www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf 15

[Hea07] ——, "Implementing and Detecting a PCI Rootkit," Black Hat, Las Vegas, USA, 2007. [Online]. Available: http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf 15

[HF06] J. Halderman and E. Felten, "Lessons from the Sony CD DRM episode," in *USENIX Security Symposium*, 2006, pp. 77–92. 50

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278 80

[HK06] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176793 13

[HLSH11] Y. Han, S. Liu, X. Su, and Z. Hu, "A dynamic analysis system for Cisco IOS based on virtualization," in *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on*, 2011, pp. 330–332. 15

[HSNB13] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of USENIX Security'13*. Washington, DC: USENIX, Aug. 2013. 87

[IEE03] "IEEE-ISTO 5001 - 2003 the nexus 5001 forum standard for a global embedded processor debug interface," IEEE - Industry Standards and Technology Organization, Dec. 2003. 21

[IEE06] *IEEE 802.15.4, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, IEEE Computer Society, Jun. 2006, iSBN 0-7381-4996-9. [Online]. Available: http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf 42

[IOZ13] IOZone, "IOZone," 2013. [Online]. Available: http://www.iozone.org/ 57

[JMF12] J. Jeon, K. K. Micinski, and J. S. Foster, "Symdroid: Symbolic execution for dalvik bytecode," 2012. 14

[JTA90] "IEEE Standard Test Access Port and Boundary-Scan Architecture," 1990, iEEE Standard. 1149.1-1990. 21

[kas15] "Equation group: Questions and answers," Feb. 2015. [Online]. Available: https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf last accessed on 2015-06-14. 50

[KCC10] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855840.1855852 15, 17, 22

[KHC08] C.-F. Kao, I.-J. Huang, and H.-M. Chen, "Hardware-Software Approaches to In-Circuit Emulation for Embedded Processors," *Design Test of Computers, IEEE*, vol. 25, no. 5, pp. 462–477, 2008. 21

[Kin76] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252 13

[Kin13] Kingston, "Secure USB Flash Drives," 2013. [Online]. Available: http://www.kingston.com/us/usb/encrypted_security 66

[Kir] A. Kirchner, "Data Leak Detection in Smartphone Applications," Master thesis, Vienna University of Technology. 34

[KKBC12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254088 14, 24

[KMPS11] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_4.pdf 20

[KPY07] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: a hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, ser. WORM '07. New York, NY, USA: ACM, 2007, pp. 46–53. [Online]. Available: http://doi.acm.org/10.1145/1314389.1314399 20

[KTC+08] S. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and Implementing Malicious Hardware," in *Workshop on Large-scale Exploits and Emergent Threats*, San Francisco, USA, 2008. 15

[LA04] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86. 10, 24

[Lal15] P. Lalet, "Scanning internet-exposed modbus devices for fun & fun," Blog post, Feb. 2015. [Online]. Available: http://pierre.droids-corp.org/blog/html/2015/02/24/scanning_internet_exposed_modbus_devices_for_fun___fun.html last accessed on 2015-05-26. 2, 90

[Lan13] R. Langer, "To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve," The Langer Group, Tech. Rep., 2013. 1, 3, 89, 91

[Lar11] J. Larimer, "Beyond Autorun: Exploiting vulnerabilities with removable storage," Black Hat, Las Vegas, USA, 2011. [Online]. Available: https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf 51

[Law96] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, no. 29es, Sep. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=326350.326357 13

[Lem13] F. Lementec. (2013) vpcie: virtual pcie devices. [Online]. Available: https://github.com/texane/vpcie last accessed on 2015-06-01. 17

[Lin12] Z. Lin, "Cs 6v81-05: System security and malicious code analysis: Dynamic binary instrumentation," Jan. 2012. [Online]. Available: https://www.utdallas.edu/~zxl111930/spring2012/public/lec4.pdf last accessed on 2015-06-01. 13

[LMP11] Y. Li, J. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *Conference on Computer and Communications Security*, Chicago, USA, 2011, pp. 3–16, ISBN 978-1-4503-0948-6. 67

[LSG+10] Y.-H. Lee, Y. W. Song, R. Girme, S. Zaveri, and Y. Chen, "Replay debugging for multi-threaded embedded software," in *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, 2010, pp. 15–22. 12

[LTCS10] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 175–183. 13

[LTHC10] H. Li, D. Tong, K. Huang, and X. Cheng, "FEMU: A firmware-based emulation framework for SoC verification," in *Hardware/-Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct 2010, pp. 257–266. 17

[LUSG04] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines." in *OSDI*, vol. 4, no. 19, 2004, pp. 17–30. 17

[LV01] F. Lerda and W. Visser, "Addressing dynamic issues of program model checking," in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, ser. SPIN '01. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 80–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=380921.380931 14

[Lyo09] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. USA: Insecure, 2009. 80

[man15] *GRLIB IP Library User's Manual*, May 2015, version 1.4.1 - b4156. [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf last accessed on 2015-06-03. 78

[Max13] Maxtor, "Maxtor Basics Personal Storage 3200 virus," 2013. [Online]. Available: http://knowledge.seagate.com/articles/en_US/FAQ/205131en?language=en_GB 50

[Mel97] C. Melear, "Emulation techniques for microcontrollers," in *Wescon/97. Conference Proceedings*, 1997, pp. 532–541. 21

[MFD11] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *USENIX Security Symposium*, San Francisco, USA, 2011, pp. 17–17. [Online]. Available: http://www.usenix.org/event/sec11/tech/full_papers/Muller.pdf 65

[MGS11] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale," in *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, Aug. 2011. 20, 46

[MHM12] F. Mischkalla, D. He, and W. Mueller, "Code generation for qemu/systemc – cosimulation from sysml," Presented at MeCoES Workshop, Oct. 2012. [Online]. Available: http://adt.cs.upb.de/mecoes/MeCoES2012/08.pdf last accessed on 2015-06-01. 17

[Mih] B. Mihaila, "Introduction to rreil." [Online]. Available: https://bitbucket.org/mihaila/bindead/wiki/Introduction%20to%20RREIL last accessed on 2015-05-30. 10

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 packets over IEEE 802.15.4 networks (RFC 4944)," IETF, Tech. Rep., Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4944.txt 42

[MMP+12] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151012 14

[MYPFH11] K.-K. Ma, K. Yit Phang, J. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis*, ser. Lecture Notes in Computer Science, E. Yahav, Ed. Springer Berlin Heidelberg, 2011, vol. 6887, pp. 95–111. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23702-7_11 14

[NB14] Nils and J. Butler, "Mission mpossible," Aug. 2014. 1, 90

[NESP08] K. Nohl, D. Evans, S. Starbug, and H. Plötz, "Reverse-engineering a cryptographic RFID tag," in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA:

USENIX Association, 2008, pp. 185–193. [Online]. Available: http://portal.acm.org/citation.cfm?id=1496711.1496724 15

[NFC11] L. O. M. Nicolas Falliere and E. Chien, "W32.stuxnet dossier," Symantec, Tech. Rep., 2011. 1, 3, 89, 91

[nis14] "Framework for improving critical infrastructure cybersecurity," National Institute of Standards and Technology, Tech. Rep., Feb. 2014. 2, 90

[NS07] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746 13

[ope94] "IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices," *IEEE Std 1275-1994*, pp. i–262, 1994. 70

[osm] "OsmocomBB." [Online]. Available: http://bb.osmocom.org/trac/45

[PD] Y.-A. Perez and L. Duflot, "Can you still trust your network card?" CanSecWest 2010. 20

[PD15] M. Purdy and L. Davarzani, "The growth game-changer: How the industrial internet of things can drive progress and prosperity," Accenture, Tech. Rep., 2015. 1

[PJFMA04] N. Petroni Jr, T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *USENIX Security Symposium*, San Diego, USA, 2004. [Online]. Available: http://usenix.org/publications/library/proceedings/sec04/tech/full_papers/petroni/petroni.pdf 16, 51

[pow11] "Power.org standard for embedded power architecture platform requirements," Power.org, Tech. Rep., Apr. 2011. [Online]. Available: https://www.power.org/wp-content/uploads/2012/06/Power_ePAPR_APPROVED_v1.1.pdf last accessed on 2015-05-28. 70

[RCK+09] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic device driver synthesis with termite," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp.

73–86. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629583 17

[Red] Redwire LLC, "Econotag: MC13224V development board w/ on-board debugging," http://www.redwirellc.com/store/node/1. 42

[RKGM14] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Guardrail: A high fidelity approach to protecting hardware devices from buggy drivers," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 655–670. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541970 17

[RKS12] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: testing drivers without devices," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 279–292. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387908 17

[RSA13] RSA, "Configuring the RSA II adapter," 2013. [Online]. Available: http://www.scribd.com/doc/3507950/RSAII-Card-Installation 66

[RSCC04] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, ser. WCAE '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1275571.1275600 13

[SA06] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 419–423. 14

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.2633

[SB03] F. Sultan and A. Bohra, "Nonintrusive Remote Healing Using Backdoors," in *Workshop on Algorithms and Architectures*

*for Self-Managing Systems*, San Diego, USA, 2003. [Online]. Available: http://www.cs.rutgers.edu/~bohra/pubs/sm03.pdf 16

[SBY+08] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008. 10, 14

[Sch10] B. Schlich, "Model checking of software for microcontrollers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, pp. 36:1–36:27, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721695.1721702 22

[Sch14] B. Schneier. (2014, Jan.) Blog post. [Online]. Available: https://www.schneier.com/blog/archives/2014/01/iratemonk_nsa_e.html last accessed on 2015-06-14. 49

[Sen07] K. Sen, "Concolic testing," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 571–572. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321746 14

[SEZ09] S. Sparks, S. Embleton, and C. Zou, "A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS," in *Symposium on Information, Computer, and Communications Security*, Sydney, Australia, 2009, pp. 125–134, ISBN 978-1-60558-394-5. 16

[Sha12] A. Sharma, "A critical review of dynamic taint analysis and forward symbolic execution," Technical report, Tech. Rep., 2012. 14

[Sha14] ——, "Exploiting undefined behaviors for efficient symbolic execution," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 727–729. [Online]. Available: http://doi.acm.org/10.1145/2591062.2594450 10

[SLC10] S.-T. Shen, S.-Y. Lee, and C.-H. Chen, "Full system simulation with qemu: An approach to multi-view 3d gpu design," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 3877–3880. 17

[SMA05] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th*

*ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750 14

[spe12] "Modbus application protocol specification," Modbus Organization, Tech. Rep., Apr. 2012, version 1.1b3. 2, 90

[SSB11] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. 36, 87

[SW07] P. H. Schmitt and B. Weiß, "Inferring invariants by symbolic execution," in *Proceedings, 4th International Verification Workshop (VERIFY'07)*, ser. CEUR Workshop Proceedings, B. Beckert, Ed., vol. 259. CEUR-WS.org, 2007, pp. 195–210. 47

[SWH+15] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015. [Online]. Available: http://www.internetsociety.org/doc/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware 15, 87

[TdH08] N. Tillmann and J. de Halleux, "Pex - white box test generation for .net," in *Proc. of Tests and Proofs (TAP'08)*, ser. LNCS, vol. 4966. Prato, Italy: Springer Verlag, Apr. 2008, p. 134–153. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=81193 14

[Tho84] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984, ISSN 0001-0782. 15, 57

[Tri] A. Triulzi, "A SSH server in your NIC," PacSec 2008. 20

[Tri08] ——, "Project Maux Mk.II, I 0wn the NIC, now I want a Shell!" in *PacSec Conference*, 2008. [Online]. Available: http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf 15

[VE14] L. Vilanova and Y. Etsion. (2014) Qemu dynamic binary instrumentation. [Online]. Available: https://projects.gso.ac.upc.edu/projects/qemu-dbi/wiki last accessed on 2015-06-01. 13

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1022920129859 14

[weba] Dagger website. [Online]. Available: http://dagger.repzret.org/ last accessed on 2015-05-31. 11

[webb] Fracture. [Online]. Available: https://github.com/draperlaboratory/fracture last accessed on 2015-05-31. 11

[webc] MC-Semantics. [Online]. Available: https://github.com/trailofbits/mcsema last accessed on 2015-05-31. 11

[webd] OpenCores. [Online]. Available: http://opencores.com/projects last accessed on 2015-06-03. 78

[webe] OpenEmbedded. [Online]. Available: http://www.openembedded.org/wiki/Main_Page last accessed on 2015-06-13. 79

[webf] Poky platform builder. [Online]. Available: http://www.pokylinux.org/ last accessed on 2015-06-13. 79

[webg] Upstream Qemu + Xilinx branches. [Online]. Available: https://github.com/Xilinx/qemu last accessed on 2015-06-03. 78

[Wei12] R.-P. Weinmann, "Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks," in *Proceedings of the 6th USENIX conference on Offensive Technologies*, ser. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372399.2372402 44

[Wel] H. Welte, "Anatomy of Contemporary GSM Cellphone Hardware." 44

[Wil12] M. Williams, "ARMV8 debug and trace architectures," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, 2012, pp. 1–6. 12, 21

[WWGZ10] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *IEEE Symposium on Security and Privacy*, 2010, pp. 497–512. 20

[XBH03] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," vol. 31, no. 2. New York, NY, USA: ACM, May 2003, pp. 122–135.

[Online]. Available: http://doi.acm.org/10.1145/871656.85963312

[YLKH11] F.-C. Yang, Y.-T. Lin, C.-F. Kao, and I.-J. Huang, "An On-Chip AHB Bus Tracer With Real-Time Compression and Dynamic Multiresolution Supports for SoC," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 4, pp. 571–584, April 2011. 78

[Zab12] W. M. Zabołotny, "Development of embedded PC and FPGA based systems with virtual hardware," in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012*, R. S. Romaniuk, Ed. SPIE, oct 2012. [Online]. Available: http://dx.doi.org/10.1117/12.98187717

[ZBFB14] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Network and Distributed System Security (NDSS) Symposium*, ser. NDSS 14, Feb. 2014. 7, 19, 95

[ZKB+13] J. Zaddach, A. Kurmus, D. Balzarotti, E. O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *ACSAC 2013, 29th Annual Computer Security Applications Conference, December 9-13, 2013, New Orleans, Louisiana, USA*, New orleans, UNITED STATES, Dec. 2013. [Online]. Available: http://www.eurecom.fr/publication/4131 7, 20, 49, 96

[zyn] Zynamics bochs python instrumentation. [Online]. Available: https://github.com/zynamics/bochs-python-instrumentation last accessed on 2015-06-01. 13