BETWEEN A HARD AND A SOFT PLACE:
THE (IN)SECURE INTERPLAY OF
HARDWARE AND SOFTWARE

Ph.D. Thesis

Lucian Cojocar

Vrije Universiteit Amsterdam, 2018
The research included in this thesis was conducted at the Department of Computer Science of the Vrije Universiteit Amsterdam.

This research was supported by the Netherlands Organisation for Scientific Research through the grant NWO 628.001.005 CYBSEC “OpenSesame: opening backdoors on embedded devices”.

Copyright © 2018 by Lucian Cojocar.

ISBN 978-94-6323-798-7

The cover was designed by Lucian Cojocar.
The thesis was printed by Gildeprint.
Manuscript version: v14-3-gea85859
BETWEEN A HARD AND A SOFT PLACE:
THE (IN)SECURE INTERPLAY OF
HARDWARE AND SOFTWARE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
on gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op maandag 30 september 2019 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

LUCIAN COJOCAR

geboren te Iaşi, Roemenië
promotor: prof.dr.ir. H.J. Bos
copromotoren: dr. C. Giuffrida
dr. K. Razavi
If the doors of perception were cleansed every thing would appear to man as it is, infinite. For man has closed himself up, till he sees all things through narrow chinks of his cavern.

William Blake, The Marriage of Heaven and Hell
Acknowledgements

I want to acknowledge the people that supported me during this wonderful journey. The journey of pursuing knowledge never ends, but here I am closing one of its chapters.

First, I would like to express my gratitude towards my supervisor, Herbert. He believed in my potential as a researcher even before starting my Ph.D. journey as he supervised me during the master's degree. Under his leadership, I had great freedom in pursuing my own ideas and I also learned how to recast and reshape the sometimes vague and crazy ideas into meaningful research questions. Thanks to his unconditional encouragement and guidance, I quickly understood how important it is to prioritize quality over immediate and half-baked results. I do not have enough words to acknowledge the amount of time and diligence that he invested in providing feedback on my scientific writing. His drive for excellence has shaped me as a researcher and inspired me to further work to improve myself. Thank you, Herbert for being a great mentor during these years.

I am especially grateful and I would also like to thank my copromoters, Cristiano and Kaveh who were always there to help with fresh and great ideas, available for brainstorming and for working together. I am hopeful that this is just the beginning of new and exciting collaborations. Thanks to you and to Herbert, I felt that what I can achieve through my journey is infinite.

Next, I would like to express my gratitude to the members of my thesis committee: Alexandru Iosup, Lejla Batina, Elias Athanasopoulos, Andy Pimentel, and Simon Moore. Thank you for dedicating time to my thesis!

Naturally, this thesis contains work that I did in collaboration with several people. I would like to thank my co-authors for their patience and awesome ideas. To Aurélien, for showing me that tinkering with hardware and embedded devices could be a building block for my journey. To Jonas, Roel and Davide, for giving me the opportunity to coordinate my first scientific project. It was a privilege to work together. I would also like to thank Kostas, for supporting me in grasping new concepts and for
his deep but contagiously-exciting explanations. To Niek and Albert, for the brainstorming sessions and for supervising me while doing unconventional experiments in the Riscure’s lab. To Sanjay, for allowing me to work on a project with high impact and for giving me hope and guidance in hard times. I would also like to thank Chen for sharing the joy of discovering new challenges in the binary analysis field. In my final year, Victor and Andrei T. helped me with data and knowledge that turned out to be crucial for finishing my work.

Of course, during the working hours and sometimes even outside of working hours, office mates were there to support me. Thanks to Andrei B., Remco, Ange-llos and Natalie for contributing to a very relaxed, yet professional, atmosphere in our P4.46 cavern. A special thanks goes to Taddeüs, not only for fixing mind-bending bugs in one of the projects (in a very short amount of time), but also for his help in finishing the paper for that project.

I want to thank the rest of my colleagues from the VUSec group: Alyssa, Ben, Dennis, Enes, Erik B., Erik K., Koen, Koustubha, Manolis, Marco, Pietro, Radhesh, Sebastian and Stephan, for the meaningful conversations that we had in the past years. Relaxing with board games during borrel time and the continuous exchange of ideas, kept me fuelled in this journey. Carrying my work had also a non-technical but important component. Caroline and Mojca, our secretaries, offered their support with a lot of patience for all sorts of endeavours.

I would also like to mention my friends Radu, Silvia, Răzvan, Roxana, Alex, Oana, Cristina, Vladimir, Vlad S., Andrei, Sorin and many more, for organizing wonderful and refreshing barbecues and for the nice weekends spent together.

I also thank my friend Călin for his time in discussing similar career paths and struggles. Thank you for sharing the right music at the right time with me.

Before starting this journey I met István. It turns out that he was always there to open my eyes and to help me in understanding the world around me. Throughout this journey, he was a collaborator, a career coach, but most importantly, he is my friend. If I would not have met you, this thesis would not exist. Thank you!

I thank my parents, Luminita and Marius, for their unconditional and unlimited support and love. There are many other things that I would like to thank my parents for, but the one that stands out is that they taught me how to balance work and life.

Lastly, I would like to thank my wonderful wife, Anca, for being next to me in every single step of this journey. She showed tremendous patience and warmth even when I was coming home grumpy. Each day, she helped me to see the world in a positive light. This kept me focused and motivated in finishing this chapter of my journey.

Lucian Cojocar
Amsterdam, The Netherlands, August 2019
## Acknowledgements

### Contents

<table>
<thead>
<tr>
<th>Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publications xiii</td>
</tr>
</tbody>
</table>

## Introduction

1.1 Reverse engineering and systems’ security ................................. 2
1.2 Research questions ............................................................. 4
1.3 Contribution and outline ....................................................... 5

## Parser Identification in Embedded Systems

2.1 Introduction ................................................................. 7
2.2 Related work ............................................................... 10
2.3 Static program analysis ....................................................... 12
   2.3.1 Identification of parser characteristics .......................... 12
   2.3.2 Lifting to an intermediate language .............................. 13
   2.3.3 Features of PARC3 components .................................... 14
2.4 Training and evaluation ..................................................... 15
   2.4.1 Scoring .................................................................. 15
   2.4.2 Validation .................................................................. 16
   2.4.3 Cross validation results .......................................... 16
2.5 Case studies ................................................................. 19
   2.5.1 GPS receiver ......................................................... 19
   2.5.2 Power meter ........................................................... 19
   2.5.3 Hard disk drive ....................................................... 20
   2.5.4 Programmable logic controller ................................. 21
2.6 Future work .............................................................. 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>Conclusion</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>A Binary Solution for Switch-Case Recovery</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>The problem with patterns</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Jump tables in practice</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Pattern matching limitations</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Tailored value set analysis for solving indirect jumps</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Detailed analysis results</td>
<td>38</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Comparing JTR with other solutions</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Related work</td>
<td>42</td>
</tr>
<tr>
<td>3.6</td>
<td>Limitations</td>
<td>43</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusion</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>Instruction Duplication</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Motivation</td>
<td>46</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Contribution</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Background</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>Related work</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Fault injection preliminaries</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Fault injection background</td>
<td>49</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Experimental fault injection setup</td>
<td>49</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Fault injection characterization</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Fault injection effectiveness</td>
<td>51</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Inaccuracies in the fault injection model</td>
<td>51</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Impact of compiler techniques</td>
<td>52</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Case study: DFA attack on software AES-128</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>SCA of ID and infection countermeasures</td>
<td>56</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Information-theoretic evaluation of ID for SCA</td>
<td>56</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Converting infection to ID for SCA</td>
<td>57</td>
</tr>
<tr>
<td>4.7</td>
<td>Practical SCA results</td>
<td>61</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Horizontal exploitation using CPA</td>
<td>61</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Horizontal exploitation using templates</td>
<td>62</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>ECC Memory and Rowhammer Attacks</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Background</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1</td>
<td>DRAM organization</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Rowhammer</td>
<td>68</td>
</tr>
<tr>
<td>5.2.3</td>
<td>ECC in DRAM</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>Threat model</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of challenges</td>
<td>71</td>
</tr>
<tr>
<td>5.5</td>
<td>Challenge C1: reverse engineering ECC</td>
<td>72</td>
</tr>
</tbody>
</table>
This thesis includes research papers published in conferences as follows:


The following related research papers are not included in this dissertation:


Chapter 1

Introduction

Embedded devices are ubiquitous. For example, an average household has at least a couple of dozen of so-called smart systems. You can find these systems in your air-conditioning system, inside your washing machine and in virtually any modern home appliance. More importantly, our life depends on embedded systems: traffic lights, cars, airplanes and power plants are just some examples of critical infrastructure components that rely on such systems. In the last decade, we became aware of cyberattacks targeting these devices (e.g., Stuxnet, Triton) and in response, we improved existing defenses and designed new mechanisms aimed at thwarting the new attacks. However, because of the high number of new embedded devices reaching the market every year, we still have a lot of work to do as defenders.

The "brain" of these devices is usually a microprocessor or micro-controller (MCU), programmed to react to events and capable of making decisions that influence our lives in one way or another. An example of the sheer number of embedded systems, surrounding us and requiring protection, is found in the automotive industry. As a modern car incorporates more than 50 MCUs [1], it is unsurprising that the automotive industry represents 8% ($32B) of the market value of all MCUs, with the highest growth rate compared to other application domains [2]. The trend of an increase in the number of embedded devices also becomes clear if we consider that more than 49 billion micro-controllers are estimated to be sold in 2019 [3], which is double the number sold five years ago. In general, due to limited resource availability, embedded devices are programmed in languages, such as C or C++, that yield very fast and compact binary code. Unfortunately, these languages offer little or no memory safety guarantees. With such a large amount of embedded devices and code (a single modern car runs up to 10 million lines of codes [4]), bugs and security bugs are inevitable. For example, the Linux kernel (another large piece of software written in C) reported more than 280 [5] vulnerabilities with serious security implications [6] in 2017, yielding 0.014 security bugs that were detected per one thousand lines of code. To put it all together, we estimate that embedded devices are susceptible to more than one million new security vulnerabilities each year.

\[\text{Common Vulnerability Scoring System (CVSS)} \geq 7\]
Motivation

One million new security vulnerabilities each year is, in fact, an optimistic estimate! This is because we based our calculation on a project (i.e., the Linux kernel) for which source code is publicly available. As opposed to closed source, on open source code virtually anyone can perform a security audit. This allows more security researchers to audit and search for vulnerabilities on open source software than on closed source one. Unfortunately for security researchers and practitioners, hardware and embedded devices are closed by nature. This means that no source is available but also that details about the inner workings of these devices are rarely public. In turn, on hardware, the security audit can be only carried by very few privileged parties. Because of lack of manpower, the quality of the audit may suffer and may leave undiscovered vulnerabilities in the final product. The only feasible choice of an external security analyst is to use reverse engineering to gain knowledge in the closed landscape of hardware and embedded devices, essentially “opening” it.

But how easy is it to reverse engineer hardware? We already know that reverse engineering is hard for binary code for which source code is unavailable. At a first glance, hardware reverse engineering is even more difficult due to additional challenges. For example, these challenges include the need of special equipment, dealing with real-time processes and events, and bypassing various tampering-proof mechanisms [6]. Compared to software reverse engineering, a different set of tools and skills are needed when reverse engineering hardware or embedded devices. This skill set is perceived as black-magic [7]. For example, anyone can run objdump on a binary but fewer analysts are willing to probe memory signals with a logic analyzer. Therefore hardware reverse engineering is evolving slower if we compare it to its software counterpart.

As we shall see, the knowledge gathered through reverse engineering techniques may improve the security of embedded devices and hardware in general. In this thesis we propose novel reverse engineering techniques for hardware and for firmware, and explore reverse engineering for security purposes.

1.1 Reverse engineering and systems’ security

To design an effective defense, one must first understand the target that needs to be protected. The better the understanding is, the more effective the proposed defense will be.

In practice, both attackers and defenders use reverse engineering (RE) techniques to increase the understanding of a system. We note that while the techniques and RE methods are similar, the main difference between the two actors is in the way they leverage the results of RE. The attacker will maliciously exploit a new vulnerability while the defender will try to fix the same vulnerability and defend against the new attack. Thus RE, by increasing the understanding of a system, also improves the

---

2 A tool that displays the binary code as assembly instructions.
3 A physical tool that captures and displays digital signals.
overall security of that system. So even system designers should reverse engineer part of their own products to improve their security.

Dealing with and defending against security vulnerabilities is not an easy task for system designers. For practical reasons, a set of security related assumptions are made during the initial design phase. For example, “Will the device be connected to a public network?”, “Is the physical integrity of this device guaranteed?” and “Who are the actors that should access the resources of this device?” are all questions to be answered before designing and incorporating any defenses in the new system. As any defense has an associated cost, be it in terms of deployment, maintenance or resource consumption, often a balance between security and cost is sought. In other words, how much are you willing to pay for a set of security guarantees under a set assumptions? These sets of assumptions along with a threat model are distilled from the perspective of an attacker and are loosely called the attack model.

In the next paragraphs we discuss several reasons why this attack model is incomplete and outdated by the time the device reaches its peak adoption. The shortcomings of the attack model stem from a poor understanding of how the components of the embedded system, i.e., its firmware and hardware, interact with each other and with the real world. As we will see in the remainder of this section, system developers can use reverse engineering to understand this interaction and improve their product design accordingly.

First, the attack model is decided usually at the start of the development process or in the development process, months before the device reaches the market. While the device is deployed and in use, new attacks or classes of attack may emerge. For example, attacks based on fault injection (i.e., clock/power glitching or inducing errors through electromagnetic pulses) were not in the attack model of early versions of smart cards. Only later it became common-practice for manufacturers to incorporate defenses against such attacks. Through reverse engineering, researchers analyzed the behavior of these devices under faults. They modeled the fault influence upon the execution of instructions. Without access to the design documents of the microcontrollers used in smart cards, researchers observed what instructions (or classes of) are susceptible to faults. This served as a starting point for designing both hardware and software defenses against this new attack class.

Second, modifying or updating the device with defenses against a new attack model is not always possible or desired in the field. A good example is the recent Rowhammer attack where silicon manufacturers are fixing the root cause of the problem in hardware. However, because these fixes cannot be deployed without changing the hardware, software defenses (albeit less secure), deliverable via software updates, may be preferred. Furthermore, a device that operates in a remote environment (e.g., a weather station, a satellite or crop sensors) is prohibitively expensive to update. Device manufacturers can leverage reverse engineering to learn the (largely undocumented) error correction function that is responsible to correct bit errors in memory. Once the guarantees of this function are known, the device manufacturer can make an informed decision whether or not Rowhammer mandates the replacement of the field device.

---

4We detail this process in Chapter 5.
Last, embedded systems are built by integrating components and systems from third parties. These components usually undergo a security review in the design phase of the system. Tracking their security guarantees throughout the life-cycle of the device is often expensive and impractical. If a vulnerability is discovered in one of these components, then the full attack model can change: now the vulnerable component once trusted becomes a threat. As an example ATMs are using storage devices (i.e., SSD or HDD) that are potentially considered trusted components because the data confidentiality is ensured by encryption. A bug in the firmware code of these devices can be converted to a back-door [8], invalidating the trust in these components. In this case, through binary reverse engineering of the device’s firmware, the security analyst can answer several questions. Does the firmware uses a strong encryption scheme to store the data on the disk? Are there any (obvious) security bugs reachable through the interface that the firmware exposes?

While the attack model is beneficial in choosing a practical and pertinent defense, it tends to grow progressively incomplete as the system ages in the field. Thus, the security of such devices deteriorates in time. Discovering security vulnerabilities at an early stage in the life cycle of a product, alleviates this problem, allowing more time for product manufacturers to secure their products.

With respect to reverse engineering we can draw two conclusions in this context. First, reverse engineering speeds up the aging process of the attack model, accelerating the bug discovery process. Second, reverse engineering can be used to discover security flaws in adjacent systems that are part of the new product. In short, reverse engineering is a powerful defense measure that should be applied even by security researchers affiliated with the product designers.

1.2 Research questions

In this thesis, we focus on reverse engineering both of the software (firmware) and of the hardware, and show that knowledge built in this way is highly relevant for the security of embedded devices and computer systems in general.

Considering that various reverse engineering techniques and methods have different security goals, costs and ease of deployment, the main research question of this thesis is:

- How can we detect vulnerabilities and uncover security-relevant properties in an embedded system through reverse engineering?

With respect to the types of reverse engineering techniques, we can further refine the main research question in two sub-questions that we answer in the following chapters.

Q1 How effective is static analysis against firmware code? (A) Can we find vulnerabilities or interesting code with a lightweight static analysis? (B) How do we improve static analysis methods to analyze the complicated control flow transfers present in firmware code? (discussed in Chapter 2 and in Chapter 3)

Q2 How can the reverse engineering of the hardware security mechanisms and reliability mechanism improve the overall security of the device? (discussed in Chapter 4 and in Chapter 5)
1.3. CONTRIBUTION AND OUTLINE

An attacker who targets an embedded device will first try to reproduce the attack on their own setup and then move to the real target. As this thesis is partly motivated by the observation that attack models are often obsolete, we will address these questions with different attack models. These attack models range, for instance, from assuming that the attacker has physical access to the target, to much less powerful models in which the attack runs remotely and stealthy.

Specifically, for Q1, we assume that the attacker can extract and only statically analyze the firmware of the targeted embedded device without them investing effort to emulate the firmware or to get access to a target device. The goal here is to find a vulnerability in the firmware that can be exploited at a later stage on a live production system. Q2 assumes a more determined attacker that has access to a similar hardware implementation. The attacker can use knowledge gathered through reverse engineering to launch the final attack. They can attack locally or, as we shall see in this thesis, they can adapt this knowledge to attack a system to which they have no physical access. This forces the defender to disavow any security through obscurity mechanism.

1.3 Contribution and outline

We argue that reverse engineering methods such as binary code analysis and non-invasive hardware analysis, are valuable tools to show shortcomings and inadequacies of the assumed security model. While reverse engineering is highly detail-oriented, in this work we focus on concepts and methodologies. We also automate reverse engineering and, where applicable, open source our proof-of-concepts and additional software.

In Chapter 2, we tackle Q1 and in particular Q1.A and propose a lightweight static analysis that uses a binary-to-LLVM lifter to find “interesting” code constructs. Embedded devices interact through inputs and outputs with the external world via several interfaces that have various levels of operational complexity. For the firmware to react to these inputs it must first decode and parse these messages because inputs are often packed in events or messages. Generating correct and bug-free parsing code, while a difficult problem in general, is a particularly interesting problem for the security of embedded devices. Because parsing code deals with untrusted data coming from the (untrusted) outside world, it represents an attractive attack surface — a vulnerability in this code has high chances to be exploitable. Our analysis (PIE) first statically detects code constructs such as switch statements, recursive functions, switch statements inside loops, etc., and then flags parsing code by computing scores according to novel metrics. We test PIE on the binary firmware of two security relevant devices: a programmable logic controller (PLC) and the firmware of a hard disk drive. In addition to parsing code, we also discover an undisclosed and parsing related vulnerability that is exposed to the outside world.

In Chapter 3, we answer Q1 and in particular Q1.B and apply and extend a type of heavyweight static analysis to firmware code. Because memory storage is limited in the embedded world, the space that a binary code takes is highly optimized. For this, a compiler is incentivised to lower the plain C switch statement to jump tables,
yielding complicated indirect control flows in the generated binary code. As we show in the chapter, these indirect flows are often missed or wrongly resolved by state of the art reverse engineering tools. Our proposed technique (named JTR) solves these cases in a compiler-agnostic manner on binary code. We analyze the firmware of a GPS module, a GSM modem and a smart meter and, in addition, we uncover an undisclosed vulnerability rooted in these intricate flow transfers.

In the remainder of the thesis (Chapter 4 and Chapter 5), we address Q2 by employing reverse engineering in two different directions. First, we infer the security details of the hardware (as well as possible attacks) by applying lightweight firmware analysis. Second, we infer details about the hardware by non-intrusive reverse engineering of the hardware itself.

In Chapter 4 we show that instruction duplication, a compiler-assisted software defense against hardware fault injection, is ineffective. Worse still, when instruction duplication is deployed, the side channel leakage through power consumption is amplified. This amplification facilitates secret information extraction from the device. Through reverse engineering of the firmware, we can assess if the instruction duplication defense is deployed and also predict what hardware resources (e.g., caches and prefetchers) are enabled on the targeted embedded device. As we show in the chapter, the protection against faults offered by instruction duplication and other defenses based on instruction redundancy, depends on the state of these hardware resources. This indicates how important the reverse engineering process is for a security review.

In Chapter 5 we reverse engineer the inner working of the error correcting codes (ECC) that are implemented in advanced memory controllers for memory reliability. To this end, we propose two novel techniques: a single bit fault injection attack and a cold boot attack. The motivation for understanding ECC stems from the Rowhammer class of attacks. Rowhammer is a technique that exploits a physical property of DRAM to induce bit changes with fast and repeated memory reads. Many people consider such attacks to be impractical on ECC memory. However, the question is how much protection ECC really offers. Because the details of the ECC algorithms are not publicly documented by the CPU designers, the only choice of the security community is to speculate on the protection guarantees offered by ECC against Rowhammer-based attacks. We not only identify these guarantees through novel reverse engineering, but also show that Rowhammer-induced bit flips are still possible even when the system uses ECC memory. Furthermore, we find that the exception-based error-reporting mechanism, designed to improve the reliability of the system and handle the events caused by error correction, introduces a timing side channel. Our proof of concept end-to-end exploit demonstrates how an attacker can now leverage this side channel and mount a Rowhammer-based attack even in the presence of ECC memory. As the current trend in embedded devices and mobile platforms is to save power at the expense of correctable errors, our reverse engineering effort serves as motivation to develop new but secure ways to use ECC that enables aggressive memory power optimizations.
Embedded systems are responsible for the security and safety of modern societies, controlling the correct operation of cars and airplanes, satellites and medical equipment, military units and all critical infrastructures. Being integrated in large and complex environments, embedded systems need to support several communication protocols to interact with other devices or with their users. Interestingly, embedded software often implements protocols that deviate from their original specifications. Some are extended with additional features, while others are completely undocumented. Furthermore, embedded parsers often consist of complex C code which is optimized to improve performance and reduce size. However, this code is rarely designed with security in mind, and often lacks proper input validation, making those devices vulnerable to memory corruption attacks. Furthermore, most embedded designs are closed source and third party security evaluations are only possible by looking at the binary firmware.

In this chapter we propose a methodology to identify parsers and complex processing logic present in binary code without access to their source code or documentation. Specifically we establish and evaluate a heuristic for detecting this type of code by means of static analysis. Afterwards we demonstrate the utility of this heuristic to identify firmware components treating input, perform reverse engineering to extract protocols, and discover and analyze bugs on four widely used devices: a GPS receiver, a power meter, a hard disk drive (HDD) and a Programmable Logic Controller (PLC).

2.1 Introduction

Embedded devices are more and more present in our everyday lives. While we rely on them for our safety and security, the frequent vulnerabilities reported in the news remind us that many of these devices have been designed without security in mind. Nowadays, typical PC systems are hardened against common software vulnerabilities. Unfortunately, this is not the case for most embedded systems. For instance, in a PC, process separation is achieved through virtual memory, protection against
stack and heap based buffer overflows are commonly inserted by compilers, and exploit mitigation such as Address Space Layout Randomization (ASLR) is adopted by most operating systems. In addition, static analysis techniques for executable code have greatly evolved in the last ten years. For example, Clang’s [9] static analysis is now able to catch many common bugs (e.g., some buffer overflows) at compile time. While not perfect, these countermeasures make traditional systems more resilient against attacks. Meanwhile, compilers used to produce software for embedded devices (firmware) often lack such protection mechanisms. Runtime exploit mitigation mechanisms such as ASLR or Data Execution Prevention (DEP) are not present or provide only a fraction of the protection offered by their PC counterparts. Moreover, many countermeasures are often omitted due to constrained budgets, limited hardware resources, or lack of incentives.

Nevertheless, these systems are often connected to the Internet and exposed to the same security threats as traditional server applications [10].

In this chapter we focus on locating complex code that is driven by user input. The most common examples in this category are parsers, but we generically refer to such code as PARC\textsubscript{3} (PArser-like Routines and Complex Control-flow Code). In practice, parser components of embedded devices represent the first line of defense in charge of processing and decoding external input. The fact that they are directly exposed to possibly malicious or malformed data makes parsers critical from a security perspective [11, 12]. In addition, parsers are often implemented using complex routines and string manipulations that are themselves prone to security bugs [13].

**Complex code and parser definition.** In this chapter, we refer to input parsers in a loose sense. Distinguishing between a lexer to separate the input stream in logical tokens, and a parser to transform the token stream into an abstract syntax tree, as is customary in compiler literature is neither relevant nor practical for our purposes. In the binary the two stages are often inseparable, due to macro expansion and inlining performed during the compilation process of the firmware. Moreover, there might not have been separate stages for lexing and parsing in the software’s design in the first place. We want to identify PARC\textsubscript{3} components in firmware, i.e., all the code that processes hardware input, and takes control flow decisions based on this input. This includes, drivers, protocol parsers, and string tokenizers. In case the tokenizer and the actual parser are separated in the binary form, our approach would identify both as candidate parsers. As programs need complex control code that operates on external data for either activity, identifying such code implies the discovery of parsing code. For this reason, we use the more generic term PARC\textsubscript{3} rather than parser to refer to our target functions.

**Analyzing the security of embedded devices.** Even though it is typically feasible to extract the firmware of an embedded system (e.g., from a memory dump or an update file), it is often very difficult to perform an automated analysis of its code. First, firmware is almost invariably stripped of debugging symbols, and contain few strings – as many devices do not even have an interface to output text to users. Second, unlike applications for regular PCs, firmware images are mostly distributed as flash chip images. The hardware abstraction, operating system, application and data form a unity designed to work exactly on the hardware platform to which they are deployed.
Without knowledge of the build process, it is hard for a security analyst to even locate functional units inside this blob. From our experience, even with access to the source code and knowledge of, say, a crash location inside the firmware, it is difficult to determine the root cause of the problem. Consequently, an independent security assessment on a binary firmware image is even more difficult and time-consuming. Finally, dynamic analysis of a firmware image is hard, as the dependence on the hardware is extremely tight. Unless one has a faithful emulator for the entire embedded device (which is rarely the case), or can debug the firmware on the device, dynamic analysis is not a feasible option. The firmware code needs the whole embedded system environment, consisting of peripherals that can be accessed via I/O memory regions, as well as hardware interrupts, to exhibit the same behavior as if it was running on the embedded device.

Analyzing PARC$_3$ functions in embedded devices. The security of parsers and PARC$_3$ code is a general problem and some of the techniques we present in this chapter are not limited to embedded systems. However, several factors make the problem of analyzing PARC$_3$ code more difficult, and also more interesting, in embedded devices.

- In embedded systems, protocol stacks are often homegrown as vendors frequently reimplement even well-known protocols, such as Hypertext Transfer Protocol (HTTP) and Transmission Control Protocol (TCP), from scratch. They do so for a variety of reasons, e.g., resource constraints, licensing issues, and legacy considerations. Unfortunately, these implementations seldom undergo the same amount of scrutiny and security analysis as code that has been tried and tested in many other environments.

- The usage of lower-level programming languages like C and C++ is frequent in embedded systems. Many protocols are textual and therefore consist of text parsing, which is, surprisingly perhaps, still one of the most difficult operations to perform securely in those languages.

- The lack of clearly defined system APIs or kernel interfaces in embedded devices makes it challenging to locate the code that actually receives and processes input values. This is also made harder because of the lack of documentation and because the firmware is often monolithic. In a traditional operating system, such input is often provided by a system call or in an environment variable. On the contrary, in an embedded system it is often read from an unknown custom hardware I/O port.

In this work we therefore focus on detecting and analyzing implementations of such code in embedded systems, without the availability of the original source code or documentation.

There are many interesting use cases beyond its obvious offensive applications. For instance, we believe it is important to be able to perform third party security evaluations of embedded systems. However, manufacturers frequently do not have the incentives and resources to hire a third party. In many cases software components are integrated in larger systems (like a car) and a security evaluation may need to
be performed on the code provided by a component supplier, who may not provide any assistance. Independent security analysis is also mandatory when the original manufacturer is not trusted. For example, a code that deals with users’ input is a good place to hide ‘features’, such as undocumented commands, deviations from the protocol specifications, or even hard-coded backdoors [14].

To solve these problems, this chapter presents a novel technique to automatically discover and analyze PARC code on embedded devices, with the goal of detecting exploitable bugs, extract protocol specifications, and find hidden commands. The system we propose, named PIE for Parser-like code Identification in Embedded Systems, first translates firmware images from binary code to the Low-Level Virtual Machine (LLVM) compiler intermediate code. Based on a classifier for statically extracted features from this intermediate code, we can detect functions that contain parser components and complex code. The classifier is trained on code samples with known parser and decision code, e.g., the coreutils programs and several servers with complex protocols.

Contributions

In this chapter we propose a novel analysis methodology to discover complex code related to parsers in embedded systems. Our technique is implemented in a prototype tool, which was successfully tested on several devices. To summarize:

- we present static analysis techniques for firmware, that relies on reverse-translation to LLVM, to perform generic detection of PARC code,
- we demonstrate the effectiveness of our techniques by evaluating them on four real world devices, e.g., to extract all implemented commands from known protocol parsers (including “hidden” commands not specified in the manual), and/or detect memory related bugs in input handling code.

We intend to release all source code of PIE to the public, to be available to the research community as a base for further development.

2.2 Related work

In this section we summarize related projects and provide the basis of the analysis methods we used in our approach.

**Static binary analysis and machine code translation.** To deal with the complexity of machine instruction sets, firmware analysis is often performed by translation of the binary opcodes into a intermediate language which explicitly express all side effects of the machine instructions. Notable binary analysis frameworks are the Binary Analysis Platform (BAP) [15] and its predecessor BitBlaze [16]. However, support for non x86 architectures is limited and fixing or extending these framework would require a considerable amount of engineering effort. LLVM has been previously integrated in cross-platform dynamic analysis systems such as S²E [17] and Panda [18][19]. We therefore decided to translate our program to the LLVM [20] intermediate language, on which
we then perform our analysis. While LLVM was designed as a compiler intermediate language, its simplicity and the availability of various transformations makes it an excellent target for decompilation. For the Intel x86 instruction set, there are several translators to LLVM \cite{21, 22, 23}. Our translator is derived from RevGen \cite{21}, which now has been incorporated into S\textsuperscript{2}E \cite{17}. Various transformations and analyses have been implemented for the LLVM intermediate language, including static slicing \cite{24} (introduced by Weiser \cite{25}) and integer range analysis (proposed by Navas et al. \cite{26}).

**Symbolic and concolic exploration of binary code.** Symbolic execution is a technique that was first proposed in 1976 \cite{27}. Since then, many symbolic execution systems have been developed, including S\textsuperscript{2}E, KLEE, FuzzBALL, and JPF. However, to the best of our knowledge, S\textsuperscript{2}E is the only one which can target the ARM architecture.

Selective Symbolic Execution (S\textsuperscript{2}E) is a framework developed at EPFL that allows symbolic execution of binary code. It leverages QEMU \cite{28} to translate blocks of binary instructions to an intermediate language, which in turn are translated to LLVM \cite{20} instructions. If symbolic values are touched by the instruction block, KLEE \cite{29} then executes the LLVM code symbolically. Using the plugin interface of S\textsuperscript{2}E, one can hook into instruction translation, execution, memory access and various other events.

Concolic execution \cite{30, 31} is an optimization of symbolic execution. It uses a pair of a concrete input and a symbolic variable to represent a concolic value. We use concolic execution instead of taint tracking because concolic execution has the advantage of tracking the full data-flow history instead of a short summary encapsulated within the tainted variable. This extra information can be used to make more precise inference about the tracked data.

**Dynamic analysis of embedded devices.** Avatar \cite{32} is an open source solution to perform binary analysis on embedded systems’ firmware. It executes binary code in an instrumented emulator, but avoids emulating the whole system by forwarding I/O accesses to the embedded device, where peripheral accesses are performed. In particular, the execution of a firmware in an emulator allows us to trace all executed instructions and memory accesses.

In Firmalice \cite{33}, Shoshitaishvili et al. use a mixed approach of static 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). Our work, while using similar techniques, aims at providing a more automated way of parser detection and has different goals. We aim to identify code that performs parsing in general, and not only code that is related with authentication.

**Protocol learning.** Polyglot \cite{34} 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 \cite{35} to reverse binary file formats. In addition, it can use information from several exam-
ple input files to gain more accurate information on file fields. Prospex identifies similar protocol messages and clusters them to recover the protocol’s state machine.

**Automatic reverse engineering.** RevNIC 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 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.

**Code complexity and embedded parsers.** Code complexity metrics have been used by Pan et al. for bug classification and detection at the source code level. Cyclomatic complexity is a metric introduced by McCabe that measures code complexity. However, Shin and Williams suggests that the correlation between source code bugs and cyclomatic complexity is insignificant. New metrics have to be used if the goal is security. Research done by Chen et al. shows that implementing embedded interpreters significantly increases the attack surface of systems. The authors provide a classification of common bugs occurring in embedded interpreters, which for example include difficulties to implement interpreters correctly in unsafe languages, incorrect handling of arithmetic errors, and preventing resource exhaustion and arbitrary execution.

### 2.3 Static program analysis

The goal of the static analysis performed by PIE is to identify PARC-like parts inside firmware code—i.e., routines associated with the analysis and parsing of data. The most interesting examples of such code are parsers. We first studied parsers to identify common features. As we shall see, features based not only on control flow, but also on data flow are stronger to successfully tell such code from other code.

#### 2.3.1 Identification of parser characteristics

In this chapter we use the term PARC to describe any piece of code dedicated to consume external input and either build an internal data structure for later use, or orchestrate the execution of the proper functionality based on the input values. Such code is often referred to simply as a parser and has been extensively studied in the compiler community as a way to perform a syntactic analysis of a computer language. For a deeper understanding, we will now specifically analyze parsing in a little more detail. In practice, the distinction between lexing and processing is not always clear and the general structure of a parser can become quite difficult to model. For instance, parsers in embedded systems are often hand-written and do not strictly separate between the two stages. Even worse, often part of the software behavior begins execution before the entire input is parsed, leading to undefined internal states if the remaining input does not correspond to what was expected.

As mentioned earlier, the distinction between lexers and parsers is not interesting for this chapter. Similarly, we are not interested in the details of the parser algorithms.
To get an understanding of what a typical parser looks like in binary format, we built a dataset containing several examples of parsers built with LEX and YACC, as well as custom parsers from open-source firmware. We then compiled them to ARM machine code, and reverse translated the ARM instructions into the LLVM intermediate language as described in Section 2.3.2. We use an intermediate representation to perform both control and data flow analysis in a format independent from the underlying machine language.

A common characteristic we noticed in these examples was the presence of two distinct patterns. The first pattern consists of the loop where the input data is fetched (e.g., by processing characters of a string or retrieving stream data from a device). The second recurrent pattern comprises the decision code, which often contains many conditional branches that depend on input values. Unfortunately, it is hard to generalize these findings. Also, not all the parsers expressed these two patterns clearly. For example, a parser might be event-driven and called by an interrupt handler to process the next character, or the decision control flow might be spread over several functions—making it difficult to detect in an automated fashion.

Since it is hard to propose general rules, we decided to extract a number of simple features and use machine learning to identify code that likely belongs to parsing routines. Each feature measures certain aspects of the code. For PARC code, we are interested in code complexity, as well as in the way in which certain values influence control flow. These features are weighted and then combined to a single scalar value, which is an indicator of the function’s likeliness to contain a parser.

2.3.2 Lifting to an intermediate language

Direct static analysis of assembler code is hard because instructions tend to have side-effects. Thus tracking data flow across assembler instructions is non-trivial. For this reason, we chose to translate the machine code to an intermediate language where instructions are side-effect free.

The LLVM intermediate language is well-suited for our purpose as data and control flow are easy to extract from its Static Single Assignment (SSA) representation. Further, using a common intermediate language instead of a particular machine language makes it easier to reuse developed techniques across different instruction set architectures. State of the art frameworks for translating machine code to LLVM did not fit our purpose as they either do not support ARM or have only partial support for it.

Because S²E uses LLVM internally and is able to run ARM code, we chose it to perform the translation from machine code to LLVM. Since S²E is originally implemented as a dynamic analysis framework, which translates code on the fly, our plugin had to significantly alter the operating principle of S²E. Our solution was to use the translation functionality to progressively translate the binary, one basic block at a time, without executing the resulting code. By analyzing the recovered code, we can discover new basic blocks, similar to the process performed by a recursive disassembler.

Because the generated LLVM code still retains many of the constructs of the origi-
inal machine language, we apply a set of transformations to normalize the results, bringing it closer to a compile-time representation. The following transformations are implemented partly as passes which are run using LLVM’s “opt” utility, and partly as python scripts using llvmpy to inspect and manipulate LLVM code.

**Control Flow Normalization.** To obtain a useful control flow representation, a function recovery pass connects translated LLVM basic blocks and groups them into functions. Functions are detected based on call and return patterns. Further, jump table patterns are detected and transformed to switch statements. This transformation is very important, as switch statements are recurring patterns in state machine implementations, which are often used in parsers. Optionally, in this step, we make use of information provided by external disassemblers.

**Data Flow Normalization.** Data flow in SSA form is considerably easier to programmatically follow than data stored in global values or stack memory. This is why we wanted to convert accesses to the assembler stack and global variables to SSA form whenever possible. In a first step, we replace accesses to Qemu’s internal representation of program memory to normal LLVM load and store instructions. A second pass detects memory accesses relative to the assembler stack pointer and transforms them to SSA. This pass first analyzes the assembler stack usage (by tracking the value of the stack pointer across the function), and then creates new SSA variables for every stack location referenced with a constant offset from the stack pointer inside the function.

Finally, we apply the scalarrepl standard LLVM pass, which breaks the structure data type created for the stack frame into individual variables, and the mem2reg pass, which transforms local variables to SSA form.

### 2.3.3 Features of PARC components

For the actual detection of complex and parsing code in firmware, we extracted a set of features from the control flow graph (CFG) and data flow graph (DFG) of each function.

**Looped switch statement (switch_loop).** Sequential parsers are typically implemented as a state machine. New tokens are fetched in a loop, and the next state or action is decided based on the current state and the next input token. This decision process usually involves switch statements or dispatch tables. Thus, identifying loops with switch statements or dispatch tables in their body are a good indicator for parsers, especially if the value influencing control flow inside the loop’s body in turn depends on the loop’s induction variable.

**Data flow analysis on conditional statements (br_fact).** While detection of switch statements already covers a large portion of parsers, compilers can choose to lower switch statements to conditional branches, or hand-coded parsers use conditional statements for example in conjunction with the strcmp function.

For this reason we analyze the influence of each variable on control flow decisions, yielding a “branching factor”. The branching factor is computed by first assigning all instructions the number 0. Then, we iterate over all conditional instructions in the function (branch, select and switch instructions). We perform a simple recursive data flow analysis on the condition value, and add the number of outgoing edges from the
2.4 TRAINING AND EVALUATION

In this section we describe how we determine the performance and relative weight of the heuristics proposed in Section 2.3.3 using regular Linux applications. For our analysis, we first obtain the LLVM bitcode files with complete control flow information (by compiling popular open source software with Clang) for a data set consisting of 101 coreutils and 3 popular applications (ProFTPd, lighttpd, and bash). We manually inspected all the coreutils and labeled each target function accordingly. For the other applications, we sample the program to mark some functions as “parser-like” (as a sanity check for obvious false negatives), and exhaustively check the result returned by PIE for false positives.

2.4.1 Scoring

As a combined score for the heuristics we normalize and weight the individual scores as follows

\[
\text{score} = \sum_{f \in \text{features}} \omega_f \frac{x_f - \min(X_f)}{\max(X_f) - \min(X_f)}
\]

where \(x_f\) represents the value feature \(f\) takes for a given function, \(X_f\) is the set of values that \(f\) takes for all tested functions, and \(\omega_f\) is the weight attributed to feature...
Thus each feature is normalized to a value between 0 and 1, and the total score is bounded. The goal of this section is to find the appropriate weights and threshold $T$ such that if the score of a particular routine exceeds the threshold, we can declare it to be PARC$_3$ code. False positives ($FP$s) are functions that score above the threshold but are not PARC$_3$ code (according to manual analysis), while false negatives ($FN$s) occurs when PARC$_3$ functions score less than $T$.

### 2.4.2 Validation

We apply 2-fold cross-validation by splitting our data set in two subsets: $S_0$ and $S_1$. We perform training by running $PIE$ on $S_0$, and do the validation on $S_1$ (later we also swap the sets). We then test each possible $\omega_f$ and $T$ combination (in 0.1 increments), and for each parameter combination, we compute the minimum distance on the ROC graph from the ROC curve to the optimum ($FP = 0$ and $TP = 1$). The output of the training step is a set of $\omega_f$ and $T$ parameters, ordered by the distance from the optimum. To obtain the weights ($\omega_f$), we average the best $K\%$ results from the training step. We then compute the average of $FP$ and $TP$ for the validation set $S_1$ using the output of the training step.

### 2.4.3 Cross validation results

Table 2.1 shows the results of the cross-validation. The first row shows results with training on $S_0$ and validation on $S_1$ ($S_0 \rightarrow S_1$). The second row ($S_1 \rightarrow S_0$) shows cross-validation when the two sets are swapped. We display the output parameters, the threshold $T$ and the weights as follows: switch_loop $\omega_0$, br_fact $\omega_1$, in_edges $\omega_2$, bb_cnt $\omega_3$, call_cnt $\omega_4$ and switch $\omega_5$. We compute $FP$ and $TP$ rates using the parameters gauged in the training step. We also display the distance $D$ from the $<FP,TP>$ point to the optimal. Because we define our goal to be as close as possible to the optimal point, $D$ is used to estimate the error ($\varepsilon = |D_{S_0 \rightarrow S_1} - D_{S_1 \rightarrow S_0}|$) of our method. Figure 2.1 shows the optimal values of the parameters using the best $K\%$ samples. Figure 2.2 displays example ROC graphs for $\omega_f$, corresponding to $K = 2\%$. The figures show that finding good, stable values for $\omega_f$ is straightforward and produces very good ROC curves.
### Table 2.1: Validation for $K = \{0.5, 1, 2, 5, 10\} \%$

<table>
<thead>
<tr>
<th>$K$</th>
<th>Direction</th>
<th>$T$</th>
<th>$\omega_0$</th>
<th>$\omega_1$</th>
<th>$\omega_2$</th>
<th>$\omega_3$</th>
<th>$\omega_4$</th>
<th>$\omega_5$</th>
<th>$FP$</th>
<th>$TP$</th>
<th>$D$</th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5%</td>
<td>$S_0 \rightarrow S_1$</td>
<td>0.234</td>
<td>0.708</td>
<td>0.483</td>
<td>0.278</td>
<td>0.349</td>
<td>0.123</td>
<td>0.691</td>
<td>0.032</td>
<td>0.982</td>
<td>0.037</td>
<td>0.0178</td>
</tr>
<tr>
<td></td>
<td>$S_1 \rightarrow S_0$</td>
<td>0.218</td>
<td>0.719</td>
<td>0.566</td>
<td>0.295</td>
<td>0.473</td>
<td>0.119</td>
<td>0.675</td>
<td>0.016</td>
<td>0.990</td>
<td>0.019</td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>$S_0 \rightarrow S_1$</td>
<td>0.244</td>
<td>0.696</td>
<td>0.496</td>
<td>0.363</td>
<td>0.353</td>
<td>0.122</td>
<td>0.689</td>
<td>0.041</td>
<td>0.996</td>
<td>0.041</td>
<td>0.0223</td>
</tr>
<tr>
<td></td>
<td>$S_1 \rightarrow S_0$</td>
<td>0.220</td>
<td>0.709</td>
<td>0.545</td>
<td>0.300</td>
<td>0.482</td>
<td>0.122</td>
<td>0.691</td>
<td>0.016</td>
<td>0.990</td>
<td>0.019</td>
<td></td>
</tr>
<tr>
<td>2%</td>
<td>$S_0 \rightarrow S_1$</td>
<td>0.247</td>
<td>0.695</td>
<td>0.502</td>
<td>0.364</td>
<td>0.413</td>
<td>0.134</td>
<td>0.685</td>
<td>0.041</td>
<td>0.996</td>
<td>0.041</td>
<td>0.0223</td>
</tr>
<tr>
<td></td>
<td>$S_1 \rightarrow S_0$</td>
<td>0.221</td>
<td>0.704</td>
<td>0.534</td>
<td>0.340</td>
<td>0.480</td>
<td>0.123</td>
<td>0.685</td>
<td>0.016</td>
<td>0.990</td>
<td>0.019</td>
<td></td>
</tr>
<tr>
<td>5%</td>
<td>$S_0 \rightarrow S_1$</td>
<td>0.263</td>
<td>0.687</td>
<td>0.502</td>
<td>0.412</td>
<td>0.515</td>
<td>0.138</td>
<td>0.670</td>
<td>0.032</td>
<td>0.987</td>
<td>0.035</td>
<td>0.0122</td>
</tr>
<tr>
<td></td>
<td>$S_1 \rightarrow S_0$</td>
<td>0.234</td>
<td>0.686</td>
<td>0.521</td>
<td>0.453</td>
<td>0.511</td>
<td>0.129</td>
<td>0.673</td>
<td>0.020</td>
<td>0.990</td>
<td>0.022</td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td>$S_0 \rightarrow S_1$</td>
<td>0.282</td>
<td>0.642</td>
<td>0.512</td>
<td>0.443</td>
<td>0.547</td>
<td>0.161</td>
<td>0.625</td>
<td>0.034</td>
<td>0.964</td>
<td>0.050</td>
<td>0.0272</td>
</tr>
<tr>
<td></td>
<td>$S_1 \rightarrow S_0$</td>
<td>0.239</td>
<td>0.618</td>
<td>0.514</td>
<td>0.529</td>
<td>0.509</td>
<td>0.134</td>
<td>0.605</td>
<td>0.020</td>
<td>0.990</td>
<td>0.022</td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.1: Best parameters for the complete data set.

Figure 2.2: ROC plot using $\omega_f$ corresponding to $K = 2\%$. 
2.5 Case studies

To show the use of PIE on real word embedded devices, we applied the tool to four case studies: a GPS receiver, a power meter, a hard disk drive, and a PLC. All four devices use SoCs or MCUs that rely on an ARM CPU core. Table 2.2 shows the complexity of their firmware in terms of number of basic blocks, functions, and call graph (CG) edges, as counted on the output of the LLVM translator. For all test cases we use PIE to select interesting (PARC$_3$) functions and briefly discuss them. For the HDD and the PLC we use the output of PIE as a basis for starting a more advanced analysis to demonstrate PIE’s usefulness from the security perspective.

2.5.1 GPS receiver

In the first experiment we analyzed the firmware of a USB GPS receiver stick. The device has a “boot loader mode”, where it receives a binary over an emulated serial port interface on the USB connection, and subsequently executes it. Using SirFDemo and SirFFlash utilities one can interact with the device and read and update the firmware. This first experiment intends to show that PIE is effective in selecting parser related functions and complex functions.

**PIE results.** For the values corresponding to $K = 2\%$ and $T = 0.247$, 2.3% of functions were marked as PARC$_3$ code and the false positive rate is 0.047. Apart from common `sprintf` and `scanf` functions which were correctly recognized as containing an input parser, PIE found a function which applies the Viterbi algorithm. Viterbi algorithm is used for signal processing to decode noisy signals and consists of complex code. Loosely, it can be viewed as a parser of the data provided by the GPS receiver. Another function automatically detected by our system parses data which seems to be in the Motorola SREC format. This is to be expected as the device’s updates are in SREC format. PIE discovered parts of the SiRF III protocol as well which was confirmed by reverse engineering. The SiRF protocol is a binary protocol, for which simple heuristics like strings search are not effective.

2.5.2 Power meter

In our second experiment we tested a remotely controlled electric energy metering device, also commonly referred to as a smart meter. The power meter contains, among other components, a GSM/GPRS modem and an infrared interface used to program and calibrate the meter.

**PIE results.** The false positive rate is 0.046 and 3.79% of functions were marked as PARC$_3$ code. Our tool automatically identified several functions responsible for string processing, similar to `scanf` and `sprintf`. In all cases, the format specifier is parsed in order to know what type of data needs to be printed. PIE also found two parsers implemented with a switch table over constant printable characters. We believe that this is the function that deals with the infrared interface of the power meter, since the switch seems to correspond to the ISO-IEC62056-21 protocol—mentioned in the manual of the device. PIE also identified the GSM/GPRS modem command handler.
Table 2.2: Firmware sizes.

<table>
<thead>
<tr>
<th></th>
<th>GPS</th>
<th>Meter</th>
<th>HDD</th>
<th>PLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (kB)</td>
<td>519</td>
<td>544</td>
<td>287</td>
<td>9,984</td>
</tr>
<tr>
<td># Basic blocks</td>
<td>48,738</td>
<td>55,054</td>
<td>43,866</td>
<td>657,349</td>
</tr>
<tr>
<td># Functions</td>
<td>1,098</td>
<td>2,332</td>
<td>6,338</td>
<td>15,437</td>
</tr>
<tr>
<td># CG edges</td>
<td>1,299</td>
<td>3,373</td>
<td>3,152</td>
<td>24,945</td>
</tr>
</tbody>
</table>

2.5.3 Hard disk drive

In our third experiment, we analyzed a commercial off-the-shelf ARM-based hard drive. After a discussion PIE’s results, we show how the output of our tool can be used to perform dynamic analysis to recover hidden commands in one of the detected parsers, and show surprising results.

PIE results. The false positive rate is 0.197 and 1.4% of functions were marked as PARC3 code. We do not consider optimized versions of the mem* functions as PARC3 code but PIE does so. The firmware has multiple versions of these functions, hence the higher false positive rate.

On the HDD, PIE found six core parsing functions among the ten highest scoring functions. In particular, the two functions with the highest score (and well above the threshold $T$) are the parsers for a simple UART menu used for maintenance, and the parser for receiving iHex-formatted firmware updates. Interestingly, the advanced UART menu (which can be enabled from the maintenance interface) was not discovered by our system. It turns out that since each character is treated on a different invocation of the UART receive interrupt and processed directly, the state machine of the menu is distributed over several functions. The experiment shows that, while the output of the PIE contains false positives (in this case arithmetic functions) and false negatives, an analyst can quickly identify the main parsers by looking at the functions with the highest score. Considering that the entire firmware contains over 6300 functions, our tool considerably reduces the amount of manual analysis time.

Example of dynamic analysis: ATA command parser

As an example of how we can use PIE for deep analysis, we analyzed one of the parsing functions we found, the AT Attachment (ATA) command parser (the hard drive’s interface to the computer). The ATA protocol is a simple command-response protocol, where a fixed structure containing block address, access size, device, etc. is sent to the hard drive. The protocol is particularly interesting to analyze because numerous commands have been changed or deprecated since its first specification in 1994, and it is well known that vendors often implement custom commands.

We wanted to know which commands our drive supports and especially if there were any commands not described in the ATA specification. Our idea was to use concolic execution on the ATA command parser, and to discover implemented commands and command options in this fashion.
For dynamic analysis, we used the Avatar framework. We first executed the firmware, with an injected debugger stub, on the HDD. Once PIE detects a PARC point of interest, we pause execution on the device with a breakpoint and take a full snapshot of the HDD’s memory and resume execution from that snapshot in $S^2E$. Accesses to I/O memory ranges are forwarded to the HDD (which is still stopped at the breakpoint), and also recorded in a trace. With the memory snapshot, and the recorded trace, we can now replay an execution. During the replay, we make use of $S^2E$’s symbolic execution engine to explore the impact of different inputs on the parser.

We used an $S^2E$ plugin to re-execute the recorded program path and identify commands and options by marking them as symbolic values. Whenever the execution left the pre-recorded execution path, we removed the symbolic state and noted the newly discovered values to inspect them later (similar to SAGE).

This way, we successfully identified 78 ATA commands, most of which are documented in the standard. Interestingly, some commands specified in the standard were not implemented by our disk, like “Read Direct Memory Access (DMA) Queued EXT” (0x26). On the other hand, the drive implements some commands that are marked as obsolete, retired or vendor specific. Among the undocumented vendor specific commands, two are particularly interesting:

- 0x80 looks like a gateway for internal firmware commands. Sending this opcode with all other registers set to “0” corrupted our drive’s configuration to the point where we needed to re-flash the firmware.

- 0xEA checks for a magic logical block addressing (LBA) of 0x333324 (“$33”), and sets a configuration value if this constant is set.

The other three vendor specific commands, 0xFA, 0xFC and 0xFD, seem to be related to normal hard drive operations. During our analysis we made another interesting observation concerning the ATA NOP (0x00) command. If the special constant 0x7654321 is presented as LBA, a second register value is treated as sub-opcode, and another parser is invoked. Depending on this sub-opcode, several functions can be called, including firmware update functionality. As the foundation of all of our analysis and results was rooted in PIE, we believe the experiment shows that PIE is useful as a starting point for the analysis of embedded firmware.

### 2.5.4 Programmable logic controller

Our last case study concerns a PLC. Such a device is part of a Supervisory Control And Data Acquisition (SCADA) infrastructure and it is normally used to automate processes in a factory. PLCs are often embedded systems that can receive inputs from sensors, send outputs to drive actuators (e.g., motors or valves), and which are equipped with a network or field bus connection to communicate with other systems in the infrastructure. Analyzing the security of this type of device is especially interesting, as they are used inside several “critical infrastructure” fields, such as power generation, water supply, and traffic control.
The PLC had the biggest firmware with the widest range of functionality among the four firmware we analyzed. It contains a proprietary operating system, the virtual machine for interpreting ladder logic programs, a web server with OpenSSL running on top of a TCP stack, and a Remote Procedure Call (RPC) library to communicate with other SCADA components and the computer used to program the PLC. The web server and the proprietary control protocol parser are two particularly interesting targets for attackers, as they are exposed over the network.

**PIE results.** With the threshold $T = 0.247$ we do not obtain any false positive but we miss many of PARC$_3$ like functions. The firmware size of the PLC is one order of magnitude bigger than the firmware of GPS, the one of power meter, or than the samples in the training set. The accuracy of our training for $T$ is biased towards small firmware. However, the score is still usable: the false positive rate for the top 0.5% functions sorted by score is 0.023. Among the top 0.5% functions, we notice code belonging to the OpenSSL library. OpenSSL is notorious for its complex code and for the large amount of parsing operations.

As an example we list some of the functions with PARC$_3$ functionality identified by PIE:

- **calls_OMSp_serializer_parser_parser**\(^1\) (position 4 in the PIE ranking) – this function is part of the ISO-TSAP protocol of the OMS proprietary storage format. From this point it is easy to detect all the other handlers for these protocols. The implementation the ISO-TSAP has been a source of errors in the last versions of this firmware.

- **miniweb_source_MWEB_VarWriter**\(^4\) (position 5) – this function resides in the web server module and it writes values to a predefined variable in the main application. The function contains a parser for the name of the variable, the value of the variable, and the type of the variable.

- **firmware_update_check**\(^2\) (position 21) – this function checks the file format of the firmware update. A firmware update file can be uploaded via the web interface of the PLC or via a special MMC card.

- **internal_var_print**\(^2\) (position 36) – function that formats a string. The type specifier tokens are not common. We believe that this function is used to print PLC’s internal variables in file logs.

- **recursive_path_lexer**\(^2\) (position 55) – a function used by the PLC’s web server that symplifies URIs. We believe that this function is used for parameter tokenization of HTTP requests. This function is a perfect example of PARC$_3$ code that can hide bugs: it is large (756 basic blocks) and it is recursive.

- **boot_menu**\(^2\) (position 58) – as we shall see, this is an undocumented feature providing a hidden boot loader menu.

\(^1\)Function named after corresponding error messages.

\(^2\)Function does not refer to any error message. The name is given by functionality.
2.5. CASE STUDIES

Finally, we chose two points of interest for further study: a parser in the boot loader accepting commands over the serial port, and the Uniform Resource Locator (URL) handling of the embedded web server. Our motivation for analyzing the boot loader parser was to understand its purpose and find a way to inject a debugging code stub in the system without hardware intervention. The web server is obviously an interesting target, as it is reachable over the network. Moreover, a quick Shodan search reveals many PLCs which are directly exposed to the Internet.

Example of dynamic analysis: boot loader parser

A quick view of the code leading to the boot loader parser showed that it is only activated by a special sequence of bytes. To extract this sequence, we used symbolic execution to injected symbolic bytes whenever the serial port was read. By looking at the values’ constraints in the symbolic state where execution enters the parser, we directly obtained the activation string. Subsequently, we sent the string to the PLC’s serial port, and the boot loader dropped in a command-response mode. Using the same technique to inject symbolic bytes when the serial port is read, we were able to understand the binary message format. Each message is prefixed with a length field and an opcode field, followed by a payload. The last byte is a simple checksum.

By observing the triggered code and the replies from the boot loader, we could also understand the meaning of messages. There were messages for querying the hardware and boot loader version whose purpose was obvious from the reply. The other messages all trigger accesses to different peripherals of the PLC, and allow for example to toggle the LEDs used to display the PLC’s status.

Thus we assume that the boot loader command interpreter is used to test the hardware without the full firmware. However, we were not able to identify a command which would allow us to read and write arbitrary memory.

Example of dynamic analysis: HTTP request handle

The PLC’s web server is custom and serves a mix of static content, processed templates and internal values. By default, it allows starting and stopping the process’ execution as well as inspecting and modifying program variables, input and output values. If the engineer chooses to, they can also embed “user pages” in their process which show the process’ state and permit control in a visually pleasing way.

We wanted to focus on the parsing of URLs, as most data is sent to the web server via HTTP GET requests. Using the output of PIE, we quickly identified the handler for the GET request. Starting from a snapshot taken at the beginning of the parser function, we replaced the URL with a short string of symbolic values and continued execution in S2E.

We found that the parser function returns an error code, which conveniently tells us if the symbolic URL was accepted by the request handler or not. Leveraging this information, we could focus on symbolic states where the URL was accepted, which revealed some interesting parameters. For example, we found that by inserting “?SRC” in the URL of a dynamically generated page, the web server would return the page template’s source code. Most probably this parameter was used to debug web page
templates or the template engine, but it is highly questionable if such undocumented parameters should be present in a release version of the PLC.

Second, symbolic execution revealed a URL prefix which exposes a web service Application Programming Interface (API). Based on the URL, we suspect that this API might be used for controlling a PLC via an IPhone. While some of the services exposed require authentication, undocumented interfaces in a security critical device should be seen with caution. Other vendors have been known to implement hidden APIs exposing privileged operations “just for convenience” [52].

Third and last, we found a bug, resulting in a software crash, in one of the input parameters. A pseudocode representation of the vulnerable code can be seen in Listing 2.1. `strtol` parses the hexadecimal number from the parameter, and returns it as a 32-bit value. Afterwards the function proceeds to check that the parsed number does not exceed a maximum threshold, and then loads a pointer from an array of structures. Even though the function seems secure, as the maximum value of the parsed number is checked, it is not: `idx` is a signed value and can be negative. This negative index is then used later to access a structure, which causes a processor exception if no memory is mapped at the pointed address. This bug has been reported to the vendor and it is fixed in the most recent firmware version.

Listing 2.1: Negative index used in a table.

```c
void get_from_hex(char *buff_in, void **ret) {
    signed int idx; void *result;

    idx = strtol(buff_in , NULL, 16);

    // idx can be negative
    if (idx >= 50)
        result = NULL;
    else
        result = 76 * idx + 0xB44FDC;
    *ret = result;
}
```

We conclude that finding two backdoors and one bug on the PLC with the help of PIE shows that PIE is very useful for analysis from a security point of view.

2.6 Future work

PIE would benefit from a more accurate data flow analysis algorithm. Combining data flow and template matching proved to be powerful, even with the limited data flow analysis that we implemented in our prototype. Using points-to analysis and inter-procedural analysis could result in a more complete picture of the code that is analyzed. Furthermore, detecting loop indices with data flow would improve the detection of lookup tables used by parser.

The next logical step to extend our system is to fully automate the application phase. As an example, it is possible to use PIE to automatically generate white-box fuzzing test cases, similar to SAGE [49] or AFL [53]. This would provide a fully
automated system for the testing of a firmware.

We also believe that testing of embedded devices would benefit from software emulation of device peripherals. In this case, the problem is that there is no behavioral specification for a peripheral, making the process of modeling it in software a daunting reverse engineering task. This lack of specifications is reflected as well on the way the protocols are implemented. If machine-readable specifications for each protocol or feature in an embedded device were to exist, automatic validation would be a powerful tool. In this case, PIE could be extended to perform automatic validation not only for protocols but as well for the behavior of the device.

A better detection of error code paths would help PIE to provide more assistance in successive symbolic execution. While detecting error paths in normal binaries is easy (e.g., a segmentation fault is easy to detect in an operating system), detecting faulty states in embedded systems is non-trivial. Each firmware treats errors individually, which is why a more sophisticated static analysis has to be devised.

2.7 Conclusion

In this chapter we described a new method for analyzing parser-like binary code in embedded devices, which we implemented in PIE. We established simple yet effective features to detect parsers and complex handling code, and evaluated them to show their potential for parser detection in binaries. We then demonstrated the practical impact of our work on four different embedded devices. For each device, we could detect complex and custom designed parsers, greatly reducing the required manual analysis time. Our case studies show our techniques can help to address the urgent problem that we currently lack almost all knowledge about unknown protocols, hidden interfaces, and additional unspecified functionality in embedded devices. We hope that improving awareness and third party analysis will help improve trust in such devices.
Most security solutions that rely on binary rewriting assume a clean separation between code and data. Unfortunately, jump tables violate this assumption. In particular, switch statements in binary code often appear as indirect jumps with jump tables that interleave with executable code—especially on ARM architectures. Most existing rewriters and disassemblers handle jump tables in a crude manner, by means of pattern matching. However, any deviation from the pattern (e.g., slightly different instructions) leads to a mismatch.

Instead, we propose a complementary approach to “solve” jump tables and automatically find the right target addresses of the indirect jump by means of a tailored Value Set Analysis (VSA). Our approach is generic and applies to binary code without any need for source, debug symbols, or compiler generated patterns.

We benchmark our technique on a large corpus of ARM binaries, including malware and firmware. For GCC binaries, our results approach those of IDA Pro when IDA has symbols (which is generally not the case), while for Clang binaries we outperform IDA Pro with debug symbols by orders of magnitude: IDA finds 11 of 828 switch statements implemented as jump tables in SPEC, while we find 763.

3.1 Introduction

Solving indirect control flow transfers such as jump tables in a disassembler is important for many applications—from binary rewriting to reverse engineering, and from malware analysis to code complexity metrics [54, 40, 55]—because it is essential to find some parts of the CFG of a program. Unfortunately, it is also very difficult and modern disassemblers frequently get it wrong in cases where code does not follow common, easy-to-fingerprint patterns, such as handwritten assembly or malware.

Extracting a reliable CFG requires the ability to distinguish between data and code and to solve the indirect control transfers—in the sense of finding the possible targets for such transfers. Any over-approximation adds spurious edges to the CFG, while under-approximations remove legitimate edges.
Unless they can extract the CFG reliably, many binary analysis techniques either no longer work at all, or with reduced accuracy. Besides reverse engineering in general, this includes the analysis of code complexity \[54, 40, 55\] and binary control flow testing \[56, 57\]. Moreover, a reliable solution for jump tables also serves to detect the presence of custom protocol parsers \[55\].

If incorrect CFGs are a nuisance for software testers and reverse engineers, they can be downright catastrophic for binary rewriting solutions. Many software hardening approaches rely on binary rewriting \[58\] to offer security guarantees. Examples include control flow integrity (CFI) \[59, 60, 56, 57, 61, 62\], sandboxing \[63, 64, 65, 66, 67, 68, 69, 70\], static taint tracking \[71, 72, 73, 74, 75\]. An incomplete or incorrect CFG can void the security guarantees or even break legitimate software. Most binary rewriting solutions \[76, 77, 78, 58, 79\] are conservative when the CFG is incomplete, trading security guarantees for the overhead of the binary solution.

State-of-the-art disassemblers use pattern matching to solve complicated indirect control flow transfers. For instance, if a specific compiler generates a jump table to implement a switch statement in C, IDA Pro should know the precise template that the compiler will use \textit{a priori}, so that it can search for exactly this pattern in the binary. Getting it right is important, as IDA uses the resulting jump targets to continue disassembly. Changing the code, however slightly, to not fit the template, results in a misclassification of the code. In practice, we found such cases in both benign and malicious software.

In this chapter, we present a generic technique to solve indirect control transfers without pattern matching, to handle complicated cases—malware and handwritten code—for which templates are not available. We do not necessarily aim to outperform solutions based on pattern matching for “easy” cases (although we show that our solution is very competitive even for those). By means of a compiler-independent context-sensitive Value Set Analysis (VSA) tailored specifically to complicated indirect control transfers, we instead aim to help disassemblers handle complex and malicious code.

We compare our work against IDA Pro, a state-of-the-art pattern matching disassembler, and show that our analysis results are good and very robust. For instance, since IDA does not have good patterns for \texttt{Clang}, our results are orders of magnitude better for \texttt{Clang} and comparable for \texttt{GCC} even though we never embedded any compiler knowledge. In summary, our contributions are the following:

- We systematize how modern compilers implement switch statements by means of jump tables.
- We show that jump table detection by pattern matching is limited.
- We describe a context sensitive VSA suitable for recovering indirect jumps from binary code that outperforms powerful tools like IDA Pro and is compiler-independent.
- We evaluate our approach and show that it recovers complicated jump tables in binary code without access to source code or debug symbols.
3.2 The problem with patterns

Modern disassemblers commonly classify all sorts of code fragments by way of pattern matching—scanning the binary code for templates of known language constructs. For example, solutions like Jakstab and IDA Pro use well-known patterns for a variety of compilers to identify function entry points, function parameters, C++ virtual calls, switch statements, and many other constructs \[80, 81, 82, 83\]. Unfortunately, the effectiveness of pattern matching depends on the completeness and soundness of the templating for the code under analysis. For instance, Bao et al. \[84\] demonstrated the ineffectiveness of pattern matching for detecting function entry points. In general, pattern matching does not work well if the code deviates from the templates—a common phenomenon in hand-written assembly or malware.

In this section, we systematize how modern compilers implement switch statements by means of jump tables. We then show the limitations of pattern matching for identifying these jump tables.

3.2.1 Jump tables in practice

Instead of a straightforward if-then-else implementation, modern compilers frequently opt for jump tables to implement switch statements \[85\]. In practice, compilers generate three different types of jump table instances in terms of the control flow. These types are orthogonal to Cifuentes and Van Emmerik’s expressions \[86\] and cover all jump tables that implement switch statements in compiler-generated code that we encountered, across hundreds of applications, a wide range of compilers, and various architectures.

Listing 3.1: jumpSIMPL: GCC implementation of switching. An alternative implementation replaces line 1 with \texttt{subs R4, R3, #10} which changes the pattern so that IDA cannot detect it.

```
1 cmp R3, #10 // compare R3 with 10
2 ldrls PC, [PC, R3, LSL #2] // if less or =, load PC
3 // using R3<<2 as index
4 // in the jump table
5 b default
6 .word 0x20
7 .word 0x40
8 .word 0x80
9 .word 0x40
10 ...
```

\[1\] 23% of switch statements are lowered to jump tables by GCC. When compiling SPEC CPU 2006 with Clang (for ARM), 21% of the switch statements are lowered to jump tables.
Listing 3.2: jump2JUMP case. Line 7 computes the value of the target. Unlike jumpSIMPL, it uses unconditional relative jumps instead of jump tables (lines 7-11).

```assembly
1 add R1, R1, #1
2 and R3, R1, #0xFF
3 cmp R3, #0xB          // 12 cases
4 mov R1, #6
5 strb R3, [R4,#4]
6 addls PC, PC, R3, LSL #2
7 b loc_7d0c          //default case
8 b loc_7d0c          //default case
9 b loc_7c9c          //case 1
10 b loc_7ccc          //case 2-9
11 ...
```

jumpSIMPL is the most common form of jump table. It uses a register as an index in the table and computes the value of that register using the switch input value. It then loads the value of an offset from the jump table, adjusts it and adds it to the program counter. An example of this idiom is shown in Listing 3.1.

jump2JUMP represents an implementation that is slightly less common, but still widely used. It first adds an offset based on the switched value to the current Personal Computer (PC). The new PC will target another jump (forward) instruction. The offsets are not stored in code, but in the branch forward instructions. Even though it uses no jump table in the strict sense of the word, we still consider this case for our experiments, since the computation of PC represents a significant and similar hurdle for static disassemblers. Listing 3.2 shows an example of the jump2JUMP idiom.

jump2STUB, a less common implementation, makes the code to jump to a stub that takes as parameters the switched value and the jump table. The jump table is stored after the unconditional jump instruction. Listing 3.3 shows an example. While less common, we did encounter this switch statement implementation on multiple occasions in ARM Thumb code, in position independent code, and in firmware. The advantage of jump2STUB is its space efficiency—the rt_switch_stub is present only once in the binary regardless of the number of switch statements.
3.2. THE PROBLEM WITH PATTERNS

Listing 3.3: jump2STUB case. The stub uses the link register (LR) to access the jump table. The jump table contains the number of cases as the first entry. The default case is the last item in the jump table.

```assembly
1  ldrb  R3, [R4, #7]      // R3 is the index
2  adds  R0, #0x49
3  bl    rt_switch_stub    // switch 7 cases
4
5  .byte 6                 // item count
6  .byte 0x4, 0x8, 0xD, 0x12, 0x17, 0x20
7  .byte 0x1D               // default offset
8
9  rt_switch_stub:          // jt width = 8 bits
10 ldrb  R12, [LR, #-1]    // load the item cnt
11 cmp   R3, R12            // compare the index
12 ldrccb R3, [LR, R3]      // load case offset
13 ldrcsb R3, [LR, R12]     // load default offset
14 add   R12, LR, R3, LSL #1 // add the offset
15 bx    R12                // jump to target
```

JTR is generic enough to recover all three cases even though we do not embed any logic that models these three types. As we shall see, we do use them for evaluation.

3.2.2 Pattern matching limitations

Disassemblers try very hard to detect switch statements (so they know which bytes to disassemble), by matching the bytes in binary code to well-known patterns that compilers are known to generate. Any deviation from the known patterns confuses the detection. Unfortunately, it is hard to find patterns that allow jump table detection to be both sound and complete. As a result, disassemblers can easily get it wrong. Consider Listing 3.1, which shows one of the idioms generated by GCC to implement switch statements. A mere replacement of the `cmp` compare instruction with any semantically similar instruction such as `sub` breaks the pattern recognition even though the program semantics remain unchanged. State-of-the-art disassemblers such as IDA miss the modified jump table entirely and interpret all the data in lines 6–10 as instructions instead.

As shown in Table 3.1, it is quite easy to fool modern disassemblers and decompilers by deviating from such well-known patterns, but the question is whether such cases also occur in real-world code. Unfortunately, they do. For instance, the last column of Table 3.1 contains code that is generated by Clang. Moreover, Listing 3.4 displays a real-world (hand optimized) implementation of the `memcpy` function in glibc. In this example (a) both conditional and unconditional instructions compute the targets (lines 5 and 15), (b) the condition of the `add` on line 5 is determined by the `and` instruction on line 3), and (c) the target computed on line 15 depends on a value computed 11 instructions earlier. Since most disassemblers assume locality (the calculation of jump targets right before the jump), they fail to recover this case. In contrast, JTR successfully computes the possible values written to the PC on lines 5 and 15. Note that link-time optimisation (LTO) may easily inline such highly optimized code
Table 3.1: Pattern matching failures. In all cases except the baseline, IDA fails to detect the switch statement (“IDA sw”). Often, this leads to an incomplete CFG also (“IDA CFG”). JTR always recovers the correct targets of the switch statement. The last and first rows of the table is code generated by compilers.

<table>
<thead>
<tr>
<th>Code</th>
<th>IDA SW</th>
<th>IDA CFG</th>
<th>JTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>// GCC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>// default</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
| cmp R3, #11    | ✓      | ✓       | ✓   | // default
| ldrls PC, [PC,R3, LSL #2] | ✓      | ✓       | ✓   |
| b default      | ✓      | ✓       | ✓   |
| // cmp->subs   | ✓      | ✓       | ✓   |
| subs R0, R3, #11 | ✓      | ✓       | ✓   | // cmp->subs
| ldrls PC, [PC,R3, LSL #2] | ✓      | ✓       | ✓   |
| b default      | ✓      | ✓       | ✓   |
| cmp R3, #11    | ✓      | ✓       | ✓   | // PC alias
| addls R3, R3, #1 | ✓      | ✓       | ✓   |
| ldrls R0, [PC,R3, LSL #2] | ✓      | ✓       | ✓   |
| movls PC, R0   | ✓      | ✓       | ✓   |
| b default      | ✓      | ✓       | ✓   |
| cmp R3, #11    | ✓      | ✓       | ✓   | // redundant
| // cond. jump  | ✓      | ✓       | ✓   |
| bhi default    | ✓      | ✓       | ✓   |
| ldrls PC, [PC,R3, LSL #2] | ✓      | ✓       | ✓   |
| b default      | ✓      | ✓       | ✓   |
| //Clang        | ✓      | ✓       | ✓   |
| //default      | ✓      | ✓       | ✓   |
| add R0, R0, #9 | ✓      | ✓       | ✓   |
| cmp R0, #6     | ✓      | ✓       | ✓   |
| bhi default    | ✓      | ✓       | ✓   |
| lsl R0, R0, #2 | ✓      | ✓       | ✓   |
| add R1, PC, #0 | ✓      | ✓       | ✓   |
| ldr PC, [R0, R1] | ✓      | ✓       | ✓   |

in several places in a program.

As a result, the analysis generates an incomplete CFG which renders subsequent analysis techniques less effective—hurting, for instance, the strength of security measures that rely on binary rewriting. Likewise, reverse engineering the code now requires significant manual annotation and analysis. In the remainder of this chapter, we show that JTR can complement pattern matching approaches and solve these cases.
3.3 Tailored value set analysis for solving indirect jumps

As shown in Figure 3.1, our analysis starts by lifting the binary to LLVM intermediate code using a home-grown translator\(^2\), much like PIE\(^5\) and LLBT\(^87\), but slightly

\(^2\)https://github.com/cojocar/bin2llvm
more advanced. As we do not consider it a contribution of this chapter, will not discuss it further. Next, in Step 2, we apply a variety of optimizations, in particular aggressive inlining. As we will discuss later, without it LLVM does not inline some of the more intricate examples of $\text{jump2STUB}$. We now describe the main analysis steps of $\text{JTR}$—Steps 3–5 in Figure 3.1. Analogous to how bounded address tracking $[88, 81, 89]$ targets VSA $[90]$ at binaries, we compute a list of all possible values a register may contain at specific points in any program—with an emphasis on indirect control transfers.

To analyse all the indirect control transfers of interest (i.e., jump tables and complex arithmetic computations on the PC), we need only consider a program’s non-constant writes to the program counter. In Step 3 in Figure 3.1 after identifying all such indirect writes (stores) to PC, $\text{JTR}$ goes through every function containing them to determine all possible paths from the store instruction back to the start of the function. For each path, we build a set $\mathcal{C}$ that represents the specific path constraints in SMT expressions form. Specifically, we go back along the paths to discover where this value originated and stop when we encounter a memory read or the start of the function. If the memory access itself depends on an indirect memory access, we recursively trace that back also, ensuring that we handle cases where, say, the program computes a pointer $p$ by adding pointer $q$ and index $i$.

To do so, $\text{JTR}$ computes a data dependency Directed Acyclic Graph (DAG) to capture the relation between the memory pointers:

1. A node in the graph corresponds to a memory pointer access in its SMT expression form. Because the SMT formula stops when we encounter a memory read, the expression kept in non-root nodes always contains a memory read. The SMT expression captures any complex expression between nodes.

2. An edge in the graph captures the dependency between nodes. Given two nodes $p$ and $q$, an edge from $p$ to $q$ means that $p$ depends on the value pointed by $q$. In other words, to solve pointer $p$, we must compute the value pointed by $q$. In this way, the expression of $p$ can easily emulate (but is not limited to) an indirect memory load with a base (node $q$) and an offset which can be a constant or another node.

3. The root of the dependency graph represents the pointer used in the targeted indirect write and the root expression will give us the possible values of PC.

For the final step, solving the DAG, the naive approach is to invoke the Satisfiability Modulo Theories (SMT) solver for the expression of the leaves, constrained by $\mathcal{C}$. Using the obtained values, we can then subsequently load the values pointed to and solve the rest of the tree. Doing so always gives results that are an overapproximation of the real jump targets, but with a high false positive rate in case of translation imprecision.

The key observation for improving the naive solution is that the possible values of a pointer are a limited subset of all possible memory addresses and $\mathcal{C}$ and some expressions of the nodes from the DAG must have a common expression. Let $\mathcal{N}$ be the set of the expressions of all nodes in the DAG. We denote $\mathcal{M}$ the set of common non-constant expressions between $\mathcal{C}$ and $\mathcal{N}$. We now construct $\mathcal{M} \leftarrow [m_0, ..., m_k]$
as a sorted set, with $m_0$ the largest expression in the set. We define the size of an expression as the number of nodes needed to represent the expression as a tree.

We now ask the Z3 [91] SMT solver for concrete values for $m_0$ while obeying the path constraints (see Algorithm 1). Using the concrete values, we recursively solve the DAG by temporarily expanding $C$ with constraints that capture the concrete values. In the second part of Algorithm 1, we start from the leaves of the DAG and we simplify each node expression using the accumulated constraints. If a node’s expression becomes constant, we load its corresponding memory pointer, otherwise we continue with the simplified expression. If the memory pointer is invalid, we abandon $m_i$ and we move to $m_{i+1}$ and restart the process. If the expression of the root node becomes constant then we successfully solved the DAG for one value. We continue the process until all the values of $m_0$ are tested.

Algorithm 1: DAG solving

\begin{algorithm}
\begin{algorithmic}
\Require $C, \mathcal{M}$
\Procedure{Solve_DAG}{\hspace{1em}}
\For{$m \in \mathcal{M}$} \Comment{$\mathcal{M}$ is an ordered set}
\For{value $\in$ SMTSolve($m, C$)}
\State constraint $\leftarrow m \equiv \text{value}$
\State $C \leftarrow C \cup \text{constraint}$
\State rootExpr $\leftarrow$ RecursiveDAGSolve(DAG.root, $C$)
\If{isConstant(rootExpr)}
\State appendSolution(rootExpr)
\State $C \leftarrow C \setminus \text{constraint}$
\EndIf
\EndFor
\EndFor
\EndProcedure
\Procedure{RecursiveDAGSolve}{Node, $C$}
\State expressions $= \{\}$
\For{child $\leftarrow$ Node.children}
\State childExpr $\leftarrow$ RecursiveDAGSolve(child, $C$)
\State expressions $\leftarrow$ expressions $\cup$ (child, childExpr)
\EndFor
\For{child, childExpr $\leftarrow$ expressions}
\If{isConstantAndLoadable(childExpr)}
\State value $\leftarrow$ LoadPointer(childExpr)
\State constraint $\leftarrow$ childExpr $\equiv$ value
\State $C \leftarrow C \cup$ constraint
\EndIf
\EndFor
\State return simplifyExpression(Node.expr, $C$)
\EndProcedure
\end{algorithmic}
\end{algorithm}

If we explored all paths but found no solution, our analysis fails. In Section 3.4, we will see that despite its simplicity this method is quite effective in solving jump tables (and other indirect jumps).

**Recovered code preparation.** As LLVM optimizations may influence our results, we evaluated the effect of important optimizations that we applied to the lifted LLVM code. As a baseline, we used the same level of optimization as in PIE [55] which already provides common optimizations such as memory to register promotion, global value numbering, and dead code elimination. Next, we added a custom pass to re-
place the intricate control flow of the `select` instruction with a simpler if-then-else sequence. Finally, we turned on aggressive inlining.

In practice, presumably because the `select` instructions does not affect the control flow of the code of interest, we could not observe any change in the solving capabilities of `JTR`. Because `JTR` analysis is intra-procedural, aggressive inlining, improved our results overall as subtler `jump2STUB` were inlined and, in consequence, analyzed. We therefore turn on aggressive inlining in Step (2) of Figure 3.1 and in all experiments in Section 3.4.

### 3.4 Evaluation

We evaluate our solution on 109 coreutils programs compiled for ARM, 4 firmware samples, a synthetic set of 210 binaries, and the SPEC CPU 2006 test suite. We believe that this is a meaningful set to evaluate `JTR`, as it is large-scale, contains binaries generated with different (known and unknown) compilers, while SPEC is commonly used by the security community for benchmarking. We summarize the results in Table 3.2 and discuss them in detail below.

**Coreutils binaries.** It is clear that if accurate patterns are available, we cannot beat pattern matching, but we show that we are competitive still with the most important state-of-the-art disassembler. As mentioned, we intend `JTR` to complement rather than compete with traditional jump target detectors—to resolve the complicated cases that pattern matching cannot handle. Nevertheless, it is interesting to evaluate our solution by itself. To show the limitations of pattern matching and the genericity of `JTR`, we use two different compilers, namely Clang (version 3.5) and GCC (version 4.9.2). We use the debug symbols in combination with IDA to generate a “ground truth.”

In the absence of debug information, IDA recovers 77% of all the switch statements. The missing 23% are either due to failed function detection, or misinterpretation of jump tables as instructions (as is the case for each of our synthetic test programs). In contrast, `JTR` recovers 98%. However, we will compare `JTR` solely with our ground truth, so as to measure against the best of what IDA could do (when IDA has the debug symbols). We believe that comparing IDA’s results on stripped binaries, even though the results look better, is less meaningful. We run our analysis on an Intel(R) i7-3770 CPU based machine with 20GB of RAM on which `JTR` took 7 seconds on average per input binary and 755 seconds in total.

The results in Table 3.2 show that regardless of the compiler in use, `JTR` yields good results. `JTR` outperforms IDA when the Clang compiler is used. This is mainly because IDA uses a pattern that is usually generated by GCC. Specifically, the code commonly generated by Clang for a jump table is `ldr PC, [Rx, Ry]`, which is different from Listing 3.1. Moreover, `Rx` and `Ry` can be any general purpose register and the index value can reside in either. Coming up with a pattern that matches Clang’s behavior and has a low false positive rates is difficult, demonstrating the benefits of `JTR`’s generic technique.

**Results on SPEC CPU test suite.** We again compiled SPEC with both Clang and GCC. The missing cases from the SPEC benchmark are either due to compilation errors (`perlbench`, `omnetpp` and `dealII` with Clang), or to translation errors. We
Table 3.2: *JTR* results. The ratios in the last rows relate to the ground truth when available, and to the “IDA + symbols” row otherwise.

(a) Test sets for which the compiler is known.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Coreutils</th>
<th>SPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GCC</td>
<td>Clang</td>
</tr>
<tr>
<td>Input binaries</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>Ground Truth</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>IDA + symbols</td>
<td>642</td>
<td>0</td>
</tr>
<tr>
<td><em>JTR</em></td>
<td>629 (97.98%)</td>
<td>295</td>
</tr>
</tbody>
</table>

(b) Test sets comprising of binaries generated by unknown (*) or multiple compilers (†).

<table>
<thead>
<tr>
<th></th>
<th>Firmware*</th>
<th>Malware*</th>
<th>Synthetic†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input binaries</td>
<td>4</td>
<td>17</td>
<td>210</td>
</tr>
<tr>
<td>Ground Truth</td>
<td>–</td>
<td>–</td>
<td>80</td>
</tr>
<tr>
<td>IDA + symbols</td>
<td>66</td>
<td>205</td>
<td>80</td>
</tr>
<tr>
<td><em>JTR</em></td>
<td>65 (98%)</td>
<td>166 (81%)</td>
<td>80 (100%)</td>
</tr>
</tbody>
</table>

instrument the *Clang* compiler to generate the ground truth. However, due to code inlining after the instrumentation, this ground truth is an underapproximation of the number of switch statements actually generated. For the testcases compiled with *GCC*, we rely on IDA's output for the ground truth (given the debug symbols).

We observe the same behavior as in the case of coreutils: *JTR* succeeds both on *Clang* and on *GCC* and pattern matching yields poor results on SPEC with *Clang*.

To show the impact of our analysis on the quality of the CFG, for the *Clang* test set, we incorporate the recovered switch statements in IDA. Due to the 9097 new edges in the CFG discovered by *JTR*, we add a cumulated 2523 basic blocks to the CFGs. The detailed results are given in Table 3.3.

**Firmware.** Next, we evaluate *JTR* on the firmware of four different devices: a smart meter, a boot-ROM used by LPC214, a GPS stick and a GSM modem. The firmware were manually reverse engineered in IDA, no symbols were available for this test. The ground truth is represented by the manual reverse engineering process. Our translator covered 66 switch statements that were implemented with jump tables, of which *JTR* identified all but one in the unoptimised LLVM bitcode. The missing jump table is recovered when aggressive inlining is enabled. We found 4 jump2STUB switch statement implementations in this set.

**Malware.** In this experiment, we used 17 malware binaries from 7 different families: AESddos, GoARM, PnScan, Taidra, Tsunami, Elknot and LightTaidra. We manually unpacked each of the samples and then fed them to *JTR*. In practice, none of the malware samples seemed to use control flow obfuscation.

Out of the 205 switch statements identified by IDA in the translated functions, the translator *JTR* recovered 166 (81%). The main cause for this modest result is the translator: several indirect jumps are wrongly translated or completely missed, there-
fore the input LLVM code for JTR is inaccurate. Interestingly, while investigating the results on this set, we also found the bug listed in Listing 3.5. This code is hand-coded assembly and part of the memset function in uClibc. It treats the length parameter (R2) as a signed value. If R2 is interpreted as a negative number, then the value written to the PC is outside of the mapped memory. The bug was confirmed by the developers of the uClibc library.

Listing 3.5: A real-world bug found by JTR.

1  memset:
2    mov  R3, R0
3    cmp  R2, #8
4    blt  continue_true  // branch if < signed
5  ...
6  continue_true:
7    movs  R2, R2
8    moveq  PC, LR
9    rsb  R2, R2, #7
10    add  PC, PC, R2, LSL #2
11    // IDA disassembler stops here
12    nop
13    strb  R1, [R3], #1  // repeated 8 times
14  ...

Synthetic binaries. In our next experiment, we again demonstrate that our solution is compiler agnostic by running JTR on 210 binaries generated from 10 C source code files that contain switch statements or control flow based on jump table. We generate the binaries using 20 different compilers and compiler optimization levels from 6 different toolchains and IDEs.

In addition to the results reported in Table 3.2, we tested JTR against 10 cases of hand-coded assembly that mimics a jump table based implementation of switch statements. All the targets of these test cases were accurately computed. JTR successfully recovers all of the 70 switch statements generated by the various compilers and reproduced by the translator.

3.4.1 Detailed analysis results

Jump table types distribution. We show the distribution of the different types of jump table, as identified by IDA, in Table 3.4. The jumpSIMPLE type is the one that is by far the most popular on ARM, regardless of the test set. jump2STUB is rare on normal binaries but much less so in the firmware test set. In the synthetic set, we generated the jump2STUB cases by selecting Thumb mode and Cortex-M0 as the target platform. This CPU is often used in embedded devices, therefore compiler flags play an important role in evaluation of tools alike JTR. Table 3.4 shows that the performance of JTR is similar, regardless of the jump table type.

Completeness and bug finding. On the ARM processor architecture, the code transitions between ARM mode and Thumb mode by means of a jump to an odd address

with a specific instruction. Depending on the path, the address computed at runtime can be odd or even. When JTR computes the possible address value, the reported value can therefore also be either odd or even, depending on the path. The two results are essentially the same (modulo the mode) and we ignore the last bit. The computation is not ARM specific, but rather arises from the generality of the solution, as JTR explores both paths (ARM and Thumb).

As shown in Listing 3.5, JTR helps to find memory access violations. However, this is not its main objective and care must be taken when applying it naively. Specifically, because our method is (a) conservative – any pointer that fails to load on a specific path invalidates that path, and (b) intra-procedural, the false positive rate for a bug finding strategy that uses JTR naively will be high. However, one may augment JTR with model checking techniques (e.g., specify a range of values that one register can have) to reduce the false positive rate or target only a specific family of bugs, such as stack-based buffer overflows.

3.4.2 Comparing JTR with other solutions

We tried to compare JTR with a variety of other solutions.

Angr. The Angr framework \cite{angr} supports ARM architecture and uses static analysis to solve some jump table. Its public version (48998c5) does not work with switch statements implemented with jump tables \cite{switch}. Again, Angr generates an incomplete CFG, as the jump table targets are missing.

Jakstab. While JakStab \cite{jakstab} does not support ARM, we tried to compare JTR with JakStab’s public version by adapting our examples to the x86 architecture. Instead of using a switch statement, we used a table of pointer to functions. With optimizations turned off, Jakstab recovers the targeted functions. When we turn on optimizations (-O2 or -O3), its analysis fails to recover the targets.

RetDec. The Retargetable decompiler \cite{retdc} which does not use any VSA techniques, fails to retrieve targets in the absence of debug symbols. The decompiler either interprets the jump table as code or it does not reference it at all.

Radare2. Radare2’s \cite{radare2} support for switch statement implemented with jump tables is work in progress \cite{radare2}. Note, however, that the implementation is based on pattern matching and therefore will have similar issues as IDA Pro.

REV.NG. Concurrent work from Di Federico et al. \cite{revng} use VSA to analyze LLVM code to recover a complete CFG. Even so, on ARM architectures, IDA’s Jackard index on CFG matching consistently outperforms REV.NG’s. The results on SPEC show that JTR improves the quality of the CFG generated by IDA. In our experience, REV.NG performed well on simple files, but none of the configurations\footnote{We compiled SPEC with Clang and with GCC. We tried static and dynamic linking.} of SPEC binaries could currently be handled by REV.NG.
Table 3.3: IDA results on SPEC binaries compiled with Clang are depicted in the first 5 columns. The first column represents the number of switch statement as reported by Clang. We instrumented Clang to tell if a switch statement was lowered to a jump table before code inlining takes place, thus the above 100% success rate on some cases for JTR. IDA misses most of the jump tables on Clang. Column 3 and 4 show how JTR performs better than IDA on the CFG benefits from the newly discovered targets. The results for SPEC when compiled with GCC are shown in the last two columns. Here JTR performs better than IDA on the total number of edges and basic blocks. The results for SPEC when compiled with GCC are shown in the last two columns.

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Clang</th>
<th>IDA</th>
<th>JTR</th>
<th>Edges added</th>
<th>BBs added</th>
<th>IDA</th>
<th>JTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>namd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 (0.00%)</td>
<td>0 (0.00%)</td>
<td>1</td>
<td>1 (100.00%)</td>
</tr>
<tr>
<td>sphinx3</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>9 (0.139%)</td>
<td>0 (0.000%)</td>
<td>2</td>
<td>2 (100.00%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>17 (0.502%)</td>
<td>2 (0.084%)</td>
<td>3</td>
<td>3 (100.00%)</td>
</tr>
<tr>
<td>milc</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3 (150.00%)</td>
<td>3 (0.113%)</td>
<td>4</td>
<td>4 (100.00%)</td>
</tr>
<tr>
<td>sjeng</td>
<td>13</td>
<td>0</td>
<td>10</td>
<td>262 (4.248%)</td>
<td>126 (3.000%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>h264ref</td>
<td>26</td>
<td>0</td>
<td>25</td>
<td>142 (0.794%)</td>
<td>10 (0.079%)</td>
<td>17</td>
<td>17 (100.00%)</td>
</tr>
<tr>
<td>soplex</td>
<td>27</td>
<td>0</td>
<td>40</td>
<td>241 (1.641%)</td>
<td>20 (0.183%)</td>
<td>36</td>
<td>40 (111.11%)</td>
</tr>
<tr>
<td>cactusADM</td>
<td>36</td>
<td>0</td>
<td>34</td>
<td>1399 (6.819%)</td>
<td>766 (5.116%)</td>
<td>34</td>
<td>29 (85.29%)</td>
</tr>
<tr>
<td>gromacs</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>367 (1.685%)</td>
<td>30 (0.192%)</td>
<td>43</td>
<td>41 (95.35%)</td>
</tr>
<tr>
<td>calculix</td>
<td>8</td>
<td>1</td>
<td>12</td>
<td>72 (0.141%)</td>
<td>0 (0.000%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>hmmer</td>
<td>41</td>
<td>0</td>
<td>39</td>
<td>298 (2.327%)</td>
<td>60 (0.658%)</td>
<td>30</td>
<td>26 (86.67%)</td>
</tr>
<tr>
<td>wrf</td>
<td>10</td>
<td>0</td>
<td>74</td>
<td>355 (0.433%)</td>
<td>1 (0.002%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>povray</td>
<td>126</td>
<td>0</td>
<td>97</td>
<td>767 (2.153%)</td>
<td>17 (0.064%)</td>
<td>95</td>
<td>75 (78.95%)</td>
</tr>
<tr>
<td>gcc</td>
<td>361</td>
<td>0</td>
<td>320</td>
<td>3467 (2.045%)</td>
<td>1038 (0.928%)</td>
<td>484</td>
<td>419 (86.57%)</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>111</td>
<td>0</td>
<td>140</td>
<td>1459 (1.265%)</td>
<td>414 (0.444%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>gobmk</td>
<td>19</td>
<td>0</td>
<td>17</td>
<td>219 (0.666%)</td>
<td>36 (0.151%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>omnetpp</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>15</td>
<td>10 (66.67%)</td>
</tr>
</tbody>
</table>
Table 3.4: Results of \textit{JTR} on different jump table types. IDA is used to categorize the jump tables whenever possible.

<table>
<thead>
<tr>
<th>Test set</th>
<th>Total</th>
<th>jumpSIMPLE</th>
<th>jump2JUMP</th>
<th>jump2STUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coreutils-GCC</td>
<td>IDA</td>
<td>642</td>
<td>642</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>629</td>
<td>629</td>
<td>0</td>
</tr>
<tr>
<td>Coreutils-Clang</td>
<td>IDA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>295</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SPEC-GCC</td>
<td>IDA</td>
<td>655</td>
<td>655</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>573</td>
<td>573</td>
<td>0</td>
</tr>
<tr>
<td>SPEC-Clang</td>
<td>IDA</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>763</td>
<td>&gt; 11</td>
<td>N/A</td>
</tr>
<tr>
<td>Firmware</td>
<td>IDA</td>
<td>66</td>
<td>58</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>65</td>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>Malware</td>
<td>IDA</td>
<td>205</td>
<td>152</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>166</td>
<td>120</td>
<td>46</td>
</tr>
<tr>
<td>Synth. binaries</td>
<td>IDA</td>
<td>80</td>
<td>59</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>\textit{JTR}</td>
<td>77</td>
<td>56</td>
<td>21</td>
</tr>
</tbody>
</table>
3.5 Related work

**Jump tables and switch statements.** Cifuentes and Van Emmerik [86] propose a solution based on lifting the binary code to Register Transfer List (RTL) expressions. Code slicing is used to extract the expression. Next the expression are substituted until any of three known patterns are reached. The summarised patterns give enough information for recovering the possible targets of the jump table. However, the recovery of jump table’s targets fails when the expression does not match one of the known patterns. Holsti [100] shows how to recover switch-case tables’ targets when a Read-Only Memory (ROM) table is present. They use partial evaluation (e.g., run the program snippet with concrete input) to generate possible outputs. For this the state of the registers is modeled and loops are unrolled. This solution does not take into account the content of the memory and is dependant on detecting switch statement implementation patterns.

Meng and Miller [101] observe the difficulty of recovering an accurate CFG because of jump tables. They define three models for jump tables usage and populate these models by means of static analysis. We believe that these models are a form of pattern matching and that are not effective on ARM architecture, for example the jump2STUB case would require information about the where in the code the stub is. Gedich and Lazdin [102] uses the linearity property of jump tables’ contents to detect them. Their solution assumes that the position of the jump table is roughly known. Once few targets are discovered, JTR can make use of this heuristic to accelerate the full jump table discovery.

Wang et al. [103] propose a solution to find data to code references. Their solution is working only when pointers to functions are stored in the jump table, in a data section. The compiler stores offsets rather than function pointer in the jump tables used by switch statements.

**CFG recovery.** Reinbacher and Brauer [88] introduce a method based on SMT for generic control flow graph recovery. They leverage forward and backward abstract program interpretation to recover indirect jump targets. As opposed to JTR, they do not take advantage of the program’s memory contents but rather use pre- and post-conditions for program’s registers which are further refined by the algorithm process. JakStab [81] uses code inlining, abstract interpretation and local constant propagation to solve jump targets. It does not work on ARM. JakStab uses Bounded Address Tracking [88] and tracks every memory access and register assignment. Updates in the abstract domain are explicitly propagated. JakStab makes a distinction between memory regions. JTR relies on expressions and it does not need this tracking. Moreover, in JTR the case when a pointer points to an unknown region is captured by the SMT expressions rather than being explicitly accounted for.

Brumley et al. [104] proposes a decompiler that uses BAP’s VSA to recover the CFG from the tested binary. Their focus is different that JTR. JTR focuses in recovering the target of indirect control flows while Phoenix focuses on recovering high level semantics (e.g., switch statements) once the CFG is known.

**Value Set Analysis.** Balakrishnan and Reps [105] introduce a binary static analysis technique called VSA. They show both limitations and strengths of VSA when applied...
3.6. LIMITATIONS

3.6 Limitations

Path explosion. A large number of paths may exist from the targeted pointer to the start of the function. Building a dependency graph for each of them and subsequently solving it could lead to resource exhaustion. The deeper into the function’s CFG the program uses the pointer, the higher the chances of running into this problem. We find that in practice, limiting the size of each path to 5 LLVM basic blocks yields good results. To further optimize the running time, JTR solves the paths in ascending order: from the shortest to the longest.

Code discovery. The accuracy of the translator, although not a real contribution of this chapter, directly influences the results of JTR. For instance, because the translator currently uses a static view of the program, it misses jump tables that the program populates at runtime. This is not a fundamental limitation and in future work, we will fix it by feeding back the JTR results. Note that dynamic jump tables are not common in benign software, but it is not hard to imagine that future malware will make use of it, as an additional defense.

Memory layout and program correctness. JTR assumes that the input code is correct and that a memory map is available. While we can extract the memory map automatically using heuristics (e.g., read/write ordering [32]), guaranteeing the correctness of the program is hard. Conversely, JTR can be instructed to find bugs. For instance, in Listing 3.5, we show one example in which JTR finds a previously unreported bug. We are confident that we can extend JTR to find good candidates for memory violation errors.

During the analysis, we should take special care when loading pointers that point to Input/Output (IO) memory. We cannot predict the value returned by load from an IO memory. The naive solution is to ignore the memory accesses to IO memory and treat them as invalid accesses. However, doing so may have a negative impact on the true positive rate of JTR in case an IO value is used to index a jump table.

Memory aliasing. Finally, the memory accesses generated by the LLVM translator can alias. When this happens the accuracy of the expressions stored in the nodes of the graph and of the path constraints decreases. The underlying reason is that JTR does not capture the aliasing information in SMT expressions. As future work, we will leverage the alias analysis already provided by LLVM to detect these cases.

LLVM Translator. Like SecondWrite [23] and PIE [55], JTR builds on top of a binary-to-LLVM translator. The translator lifts the code to LLVM in a straightforward
manner and JTR then analyzes the resulting code together with the memory image of the binary. Specifically, it uses weak heuristics for determining whether a function is in ARM mode or Thumb mode and occasionally misclassifies them. In addition, the translator does not itself resolve the indirect jump targets and its recursive descent disassembly therefore misses code fragments. The solution for the latter problem would be to feed the results of the JTR analysis back to the translator to discover the targeted code, but doing so is a major engineering task, and we leave this for future work.

Misclassifying a fragment’s mode (ARM or Thumb) and missing code fragments in the recursive descent both cause JTR to miss indirect jumps and hence the appropriate targets. We stress that these issues are a problem of the translator only and not of the JTR analysis. By construction, JTR will generate a solution for the targets of every indirect jump in its input.

### 3.7 Conclusion

Jump tables on RISC architectures lead to frequent interleavings of (jump table) data and code in binaries. Most disassembler use pattern matching to detect such jump tables in binary code, which easily fails for complicated indirect control transfers. We argue that in specific security-relevant domains (handwritten code, firmware and malware), we need a more generic technique to handle the cases that elude common pattern matching. This chapter proposed such a technique for “solving” jump targets for indirect control transfers. By transforming the targets to formulas that we solve in an SMT solver, we remove dependencies on templates, compilers, and processor architectures. The results show that our technique approaches and sometimes improves that of popular disassemblers that use pattern matching. JTR is available as an open source project: [https://github.com/cojocar/jtr](https://github.com/cojocar/jtr)
Fault injection attacks alter the intended behavior of micro-controllers, compromising their security. These attacks can be mitigated using software countermeasures. A widely-used software-based solution to deflect fault attacks is instruction duplication and n-plication. We explore two main limitations with these approaches: first, we examine the effect of instruction duplication under fault attacks, demonstrating that as a fault tolerance mechanism, code duplication does not provide a strong protection in practice. Second, we show that instruction duplication increases side-channel leakage of sensitive code regions using a multivariate exploitation technique both in theory and in practice.

4.1 Introduction

Fault Injection (FI) and Side-Channel Analysis (SCA) attacks are a risk for micro-controllers operating in a hostile environment where attackers have physical access to the target. These attacks can break cryptographic algorithms and recover secrets either by e.g. changing the control flow of the program (FI) or by monitoring the device’s power consumption (SCA) with little or no evidence.

Multiple countermeasures such as random delays [108], masking [109], infection [110], data redundancy checks [111, 112] and instruction redundancy [113] have been proposed to tackle these threats, yet their impact, effectiveness and potential interactions remain open for investigation. Such countermeasures can be implemented at hardware or at software level, often translating to overheads in silicon area and execution runtime. This exacerbates the need for a detailed analysis of the benefits introduced by these countermeasures before their actual deployment.
4.1.1 Motivation

In this work, we focus on the Instruction Duplication (ID) countermeasure, applied as a fault tolerance mechanism in software. The assembly-level redundancy introduced by ID can prevent attacks aiming to skip instructions and alter the control flow. Recent defenses (e.g., infection [114]) build further on code redundancy in order to provide a stronger protection.

Manually applying these defenses, however, does not scale well for a large code base that needs to be protected: it is an error-prone process and it costs many highly skilled man-hours, therefore, in practice, it is often automated using compiler techniques [115][116][117]. On top of protecting against fault attacks, compilers can also provide support to reduce the information leakage through side channels [118][119][120][121][122]. While there is previous work exploring the effect of one defense mechanism on another [123][124][125], to the best of our knowledge, the effect of ID on side-channel leakage has not been explored before. We perform an in-depth investigation of ID, focusing on its applicability against FI as well on its interaction with side-channel attacks.

Specifically, regarding fault attacks, the defender needs to exercise caution when applying ID, since the device may not adhere to the “single instruction skip” model. In such cases, the countermeasure is ineffective and we demonstrate that it can even benefit certain fault injection strategies. In addition, we highlight how even an effective application of ID can enhance our capability to perform side-channel attacks on the underlying implementation. Thus, we establish that care needs to be taken with respect to the equilibrium between fault injection defenses and side-channel resistance.

In the process of investigating these software defenses, we built the first open-source compiler capable of generating duplicated code for any C/C++ program. In this way, we hope to stimulate further research in this area.

4.1.2 Contribution

We summarize our contributions as follows:

- We experimentally determine that instruction skipping is not a realistic fault model for modern ARM Cortex-M4 MCUs.
- We develop and open source an instruction duplication compiler for ARM Thumb2 architectures. To our knowledge, this is is the first time that such a compiler is publicly available.
- We examine the interaction between n-plication and side-channel resistance and demonstrate the trade-off using an information-theoretic approach. In addition, we show how horizontal exploitation techniques can leverage the side-channel introduced by ID-based defenses.
- We examine how the redundancy of infective countermeasures can interact with side-channel resistance and demonstrate how a Hidden Markov Model can render infection [114] equivalent to ID from a side-channel point-of-view.

---

1The code is available at: https://github.com/cojocar/llvm-iskip
4.2. BACKGROUND

This chapter starts with the background (in Section 4.2) and with an overview of the related work in Section 4.3. Sections 4.4 and 4.5 investigate the limitations of the assumed FI model as well as the limits of compiler-based ID. In Sections 4.6 and 4.7 we determine the impact of hardening code with ID on SCA attacks. We summarize our findings in Section 4.8.

4.2 Background

Software-based instruction redundancy methods for fault detection were proposed by Barenghi et al. [113]. In this technique, the original stream of instructions to be executed is duplicated (or even triplicated), one instruction after another, either manually or automatically [115, 126, 117].

For example a load from memory (ldm R0, [R2, #0]) is transformed by duplication in two loads originating from the same memory. To provide fault detection the destination registers must be different and then checked for differences (Listing 4.2). Under single instruction skip model, the fault tolerance arises when using the same register as destination. Indeed, skipping one single instruction from Listing 4.1 has the same effect as executing the original instruction.

Listing 4.1: Fault tolerance.
\begin{verbatim}
  ldr R0, [R2, #0]
  ldr R0, [R2, #0]
\end{verbatim}

Listing 4.2: Fault detection.
\begin{verbatim}
  ldr R0, [R2, #0]
  ldr R1, [R2, #0]
  cmp R0, R1
  bne fault_detected
\end{verbatim}

In practice, Moro et al. [116] showed that every ARM Thumb-1/2 instruction can be duplicated. We differentiate three classes of instructions: idempotent instructions, separable instructions and specific instructions. While the idempotent instructions are duplicable with no extra transformation, the other two classes often require an extra register to perform the duplication.

Therefore, on ARM Thumb-1/2, ID is generic and can be applied automatically regardless of the algorithm that the instruction stream implements.

Automatic deployment. Maebbe et al. [117] apply ID for fault detection at link-time for the ARM architecture. Barry et al. [115] described a compiler able to produce duplicated instructions, however their tool is not publicly available.

Our LLVM based compiler emits duplicated instructions for the ARM Thumb2 instruction set. Through code annotations, the hardening can be enabled or disabled at function level, as instructed by the developer. The modified LLVM based compiler has a similar architecture as the implementation described by Bary et al. [115] and it can compile code in any language supported by Clang (e.g., C, C++) with different optimization levels, including the AES-128 implementation used in this chapter. It is
designed to be a drop-in replacement for any LLVM based toolchain. Due to space constraints we omit the implementation details. The compiler is available as an open-source project.

### 4.3 Related work

**ID and the FI model.** Moro et al. [126] practically evaluates instruction duplication as a defense for FI on a Cortex-M3 microcontroller (MCU). They use electromagnetic (EMI) pulses to insert glitches and show the importance of the fault model. Riviere et al. [127] show that the single instruction model is invalid when caches are enabled. The observed skip behavior, in the presence of an EMI glitch, is: the last 4 instructions are re-executed and 4 instructions are skipped — this partially invalidates the instruction duplication defense. Dureuil et al. [128] model the fault injection attack by including the EMI probe position. When an attack succeeds, the most probable outcome is to skip 1-4 instructions on a common smart card. They show that a probable outcome is the corruption to 0 of the destination operand of a `ld` instruction. Yuce et al. [129] show the effect of a single clock glitch on the ID scheme at clock granularity. They observe that the first instance of the instruction is corrupted and that its duplicated counterpart is transformed to a NOP instruction, thus defeating the ID. They use a 7-stage FPGA based implementation and clock glitches for experiments. Instead, we use a 3-stage pipeline off-the-shelf device and voltage glitches to investigate ID.

**ID and SCA interaction.** Regazzoni et al. [130] first looked at the interaction between fault injection defenses and Power Analysis (PA) attacks. Specifically, they studied an AES implementation with parity based error detection circuitry. They conclude that the presence of a parity error detection circuit will leak important information to an attacker through PA. One year later, Regazzoni et al. [131] experimentally show the exploitability of an known-by-the-attacker error detection circuit. Pahlevanzadeh et al. [132] look at three fault detection methods designed specifically for AES: double module redundancy, parity checks, inverse execution; all implemented on an FPGA. They find that parity checks are actually improving the resistance against standard Correlation Power Analysis (CPA). Similarly, Luo et al. [133] use CPA to attack an FPGA implementation of AES which is hardened for fault detection. They conclude that duplication does not improve the success rate of the attack in respect to the un-hardened AES implementation. However, we stress that the approaches of [132, 133] use naive CPA attacks and do not rely on multivariate, horizontal exploitation of the leakage. Such attack-dependent techniques do not reveal the full picture and may lure the side-channel evaluator in a false sense of security.

### 4.4 Fault injection preliminaries

Because ID and n-plication are defenses for faults, we experimentally evaluate them in a realistic fault injection scenario.
4.4.1 Fault injection background

Fault injection attacks change the intended behavior of a target by manipulating its environmental conditions. This can be accomplished using different fault injection techniques such as: voltage FI, electromagnetic FI and optical FI. In this chapter we focus only on voltage FI where glitches are introduced in the voltage signal that powers the subsystem responsible for executing software. Voltage FI is easy to mount as it does not require sophisticated equipment and it is invasive.

FI model. Faults can target different physical layers of the device: single transistors, logic gates or computation units [134]. In this chapter, we are interested in the observable effect of faults, namely, in faults that can cause a change in the program flow and that manifest at the instruction level. We note several types of faults in respect to instructions: single instruction skip [135], multiple instruction skip [127, 136], instruction re-execution [137, 127] and instruction corruption [136]. These types of faults are from now on referred to as the fault model.

Fault injection parameters. The following glitch parameters are important when performing voltage FI:

- the Normal Voltage is the voltage supplied to the target.
- the Glitch Voltage is the voltage subtracted from the Normal Voltage when the glitch is injected.
- the Glitch Offset is the time between when the trigger is observed and when the glitch is injected.
- the Glitch Length is the time for which the Glitch Voltage is set.

Finding the right parameters for a target is defined as characterization.

4.4.2 Experimental fault injection setup

Fault injection target. All fault injection experiments described in this section are performed targeting an off-the-shelf development platform built around an STM32F407 MCU. This MCU is implemented using 90nm technology and includes an ARM Cortex-M4 core running at 168 MHz. This Cortex-M4 based MCU has an instruction cache, a data cache and a prefetch buffer. Related research used a similar experimentation target. Moro et al. [138, 126] used a development board designed around an 130nm technology MCU featuring an ARM Cortex-M3 core running at 56 MHz. The Cortex-M3 and Cortex-M4 are very similar and we expect the differences to have minimal impact. The latter includes additional specialized instructions which are not targeted in this chapter. The pipeline size (3 stages) and the rest of the instruction set are the same.

To avoid instruction re-execution, which was shown to be possible by Rivier et al. [127], all experiments are performed with the prefetch buffer disabled and with caches enabled, unless otherwise stated.
Fault injection tooling. The voltage FI test bed is created using Riscure’s VC Glitcher product that generates an arbitrary voltage signal with a pulse resolution of 2 nanoseconds. Similarly to previous work, in a synthetic setup, we use a general purpose input output (GPIO) signal to time the attack which allows us to inject a glitch at the moment the target is executing the targeted code. The target’s reset signal is used to reset the target prior to each experiment to avoid data cross-contamination.

4.4.3 Fault injection characterization

We use the code snippet from Listing 4.3 for two purposes: (a) to find the glitch parameters (characterization) and (b) to invalidate the single instruction skip model for the target described in Section 4.4.2. The code is a copy-loop construction that is known to be a common target for fault injection because it has significant duration. The targeted code is executed in a loop to minimize the impact of the Glitch Offset parameter as it does not matter what iteration of the loop and which part of the loop is hit.

Listing 4.3: Characterization code.

```assembly
loop:  ldm  R0,  {R4-R10}
       stm  R0,  {R4-R10}
       subs R1,  #1
       bne  loop  //loop back
```

The target’s susceptibility to voltage FI attack is determined using the follow-

[https://www.riscure.com/security-tools/hardware/vc-glitcher]
4.5. FAULT INJECTION EFFECTIVENESS

In this section, we practically evaluate ID under instruction corruption FI model.

4.5.1 Inaccuracies in the fault injection model

We resort to two experiments, that show how ID can negatively affect the fault tolerance of ID if a different model than single instruction skip holds. Furthermore, we show that when applying ID the runtime configuration of the target must be considered.

ID and the “real” FI model

We determine the impact of ID by duplicating and n-plicating code from Listing 4.3. For each code instance, we perform 10K experiments, using the glitch parameters outlined in Section 4.4.3.

Table 4.1 shows that ID does not provide fault tolerance for software for our target. Even if the instruction is n-plicated three times or more, the fault tolerance is not substantially improved. Because we use a real target with no access to low level
Table 4.1: Success rate of FI and n-plication levels.

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>n = 2 (ID)</th>
<th>n = 3</th>
<th>n = 4</th>
<th>n = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR (%)</td>
<td>15.91</td>
<td>15.61</td>
<td>11.59</td>
<td>13.5</td>
<td>11.96</td>
</tr>
<tr>
<td>SE ($\times 10^{-4}$)</td>
<td>25</td>
<td>25</td>
<td>22</td>
<td>24</td>
<td>22</td>
</tr>
</tbody>
</table>

hardware features (i.e., flip-flop states), we do not aim to detail the root cause of this behavior. Instead, we note that the instruction corruption model captures this result.

Limitations of a static FI model

When ID is deployed automatically at compile time, the compiler is not aware of the runtime configuration (e.g., cache configuration). In this experiment, we show how ID and n-plication affects the success of FI when several runtime configurations are used.

In Figure 4.2 we enable and disable the prefetch buffer (p), the instruction cache (i) and the data cache (d) and plot the fault injection success rate on the code similar to Listing 4.5. A capital letter in the title of the subplot means that the specific feature is enabled. We use the color scheme defined in Section 4.4.3.

Because the data on which our test operates is stored in registers, toggling the data cache has no impact on the fault tolerance. However, we observe four interesting results. First, ID increases the probability of a successful fault when the device is used with all its functionality enabled (PID). In this case, n-plication with $n = 3$ and $n = 4$ has the highest fault tolerance. Second, when all features are disabled (pid), none of the n-plication level improve the fault tolerance. Thirdly, when the instruction cache is disabled, enabling the prefetch buffer makes ID the most effective amongst the n-plication levels (pid, piD vs. Pid, PiD). Finally, comparing the right-most four subplots with the left-most subplots, the instruction cache offers an improved resilience against voltage glitches.

As a consequence, the compiler must be aware of the runtime configuration of the device when it emits redundant instructions.

4.5.2 Impact of compiler techniques

We now explore two compiler techniques that affect the effectiveness of ID.

Register allocation pressure

Register Allocation (RA) is the process in which the compiler maps the virtual (unlimited) registers to physical (limited) registers. This process is highly optimized to yield the best space and runtime performance. In this section we show that the modified register allocation scheme that ID requires has a negative impact on the fault tolerance.
4.5. FAULT INJECTION EFFECTIVENESS

Figure 4.2: SR of faults vs. multiple application levels and runtime configurations.

Listing 4.4: Registers are incremented.

```assembly
add R5, R5, #1
add R7, R7, #1
```

Listing 4.5: Code ready for duplication.

```assembly
add R4, R5, #1
mov R5, R4
add R6, R7, #1
mov R7, R6
```

Listing [4.5] is the transformation of the code from Listing [4.4] with the \texttt{add} being replaced by an idempotent sequence that uses an extra temporary register (see Section 4.2). We define a successful glitch with respect to the contents of the registers \texttt{r5} and \texttt{r7}. If the contents of the registers is different than what is expected (i.e., the number of iterations added to the initial value of the registers) then we count this trial as a success. Otherwise, the glitch was not inserted or the parameters caused a \textit{mute}.

The ID aware RA yields a higher success rate for FI (SR=18.64\%, SE=20x10^{-5}) than the unmodified one (SR=10.63\%, SE=16x10^{-5}). Apart from runtime performance degradation, the increased register pressure induced by the custom RA has a two fold negative impact on the fault tolerance. First, it increases the probability of a register to be spilled on the stack. As a consequence, the compiler will likely chose complex multi-memory access operations over simple load or stores. The multi-memory operations (e.g., \texttt{ldm}, \texttt{stm}) are more prone to faults than single memory operations \cite{136} or than register to register operations. Second, an extra instruction to write back the result is needed (\texttt{mov}). This extra instruction is duplicated, therefore it increases the window in which a fault can be injected and it adds another leakage point.

In short, not only does the ID register allocation works against the established RA
optimizations, but it also has a negative effect on the fault tolerance guarantees. This is a fundamental limitation of ID.

### Instruction ordering

The compiler has the freedom to emit instructions in any order. This is done either for optimization purposes (e.g., benefit from a multi-stage pipeline) or to avoid a certain illegal order of instructions. Barry et al. [115] showed that the correct scheduling of duplicated instructions can reduce the runtime overhead of the duplicated code, from 2.14X down to 1.70X-2.09X on a software AES implementation. Yuce et al. [129] hint at the interaction between ID and the processor pipeline.

To analyze what is the impact of the instruction order on the success rate of injected faults we compare the success rate of the code Listing 4.6 and its possible scheduled version Listing 4.7. We define a successful trial whenever the memory pointed by $r6$ is different than its initial value. Our results show that instruction scheduling decreases the success rate of injecting a fault, from 8.51% to 4.00%.

Intuitively, the pipeline for Listing 4.6 contains the protected instruction and its copy right after another. Therefore, the chances that a fault affects the protected instruction and its copy at a given clock cycle is higher than in the case when the protected instruction and its copy are one (or more) instruction apart (Listing 4.7). These results are in line with the work of Yuce et al. [129], which shows that ID can be bypassed with a single glitch because multiple instructions are in the pipeline at a given clock cycle.

When emitting duplicated code the order is important, yet to date a FI model that captures the order interaction does not exist, let alone a compiler that uses this model. We leave the design of such a model and compiler as future work. We conclude that compiler optimization techniques (e.g., instruction scheduling, register allocation optimality) interact with the fault tolerance guarantees of ID.

Listing 4.6: Natural order.

```
add R0, R4, R1
add R0, R4, R1
ldr R5, [R6, #0]
ldr R5, [R6, #0]
```

Listing 4.7: Possible re-ordering.

```
add R0, R4, R1
ldr R5, [R6, #0]
add R0, R4, R1
ldr R5, [R6, #0]
```

### 4.5.3 Case study: DFA attack on software AES-128

In previous sections we determined the impact of ID as a fault tolerance mechanism on synthetic code. Now we show the interaction between ID and the number of trials needed to conduct a fault based attack. To this extent, we automatically apply ID
on a large and complex code construction, the AES-128 cryptographic algorithm, and perform the Differential Fault Analysis (DFA) attack described by Dusart et al. \cite{139}. The goal of the attack is to extract the fixed key by observing the faulty output.

We use the tiny-AES128-C\footnote{https://github.com/kokke/tiny-AES128-C} implementation of the AES-128 cipher, in ECB mode for our target to encrypt a fixed input with a fixed key. A trigger is implemented between the 9\textsuperscript{th} and the 10\textsuperscript{th} round to guarantee we always hit the right location within the algorithm. Two versions of the AES-128 implementation are compiled: a hardened version (with ID in place) and an non-hardened version.

A 2K trace set containing traces with faulty outputs is acquired for each implementation. We randomly select $n_t$ from these trace sets and use them in the DFA attack. We repeat this process 100 times for each implementation and we plot how often the attack is successful in Figure 4.3.

![Figure 4.3: DFA on AES-128.](image)

The non-hardened implementation outperforms the hardened implementation in terms of FI tolerance. A clear indication that ID is not effective for protecting the AES-128 algorithm when the instruction corruption fault model holds. Depending on the time penalty required for a single experiment, the small difference can have a noticeable effect. If the target needs to be reset before each experiment then tens of seconds are added for each experiment. Moreover, the target might remove or change the keys after a limited amount of encryptions.

We analyzed the outputs in more detail and counted how often multi byte changes are observed in both implementations (Table \ref{table:multi-byte-changes}). From the number of all faults observed (i.e., at least 1 byte difference), 4 bytes faults\footnote{These are the faults useful for DFA on AES} are more probable to be observed in the hardened implementation.

To conclude, fewer successful faults are needed to attack the hardened AES.
Table 4.2: Bytes changed in the output.

<table>
<thead>
<tr>
<th></th>
<th>3 or less</th>
<th>4</th>
<th>5 or more</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardened</td>
<td>0.2%</td>
<td>64.0%</td>
<td>35.7%</td>
</tr>
<tr>
<td>Unhardened</td>
<td>1.1%</td>
<td>41.5%</td>
<td>57.4%</td>
</tr>
</tbody>
</table>

### 4.6 SCA of ID and Infection Countermeasures

This section demonstrates the interactions between the redundancy-based FI countermeasures and the side-channel resistance of an implementation that is employing them. In Section 4.6.1, we analyze the theoretical effect of ID and n-plication on SCA using an information-theoretic approach. Section 4.6.2 demonstrates how to perform SCA on infective countermeasures using a Hidden Markov Model that simplifies the exploitation phase of infection to that of ID. Throughout this section, capital letters denote random variables and small case letters denote instances of random variables or constants. Bold letters denote vectors.

#### 4.6.1 Information-theoretic Evaluation of ID for SCA

From a side-channel perspective, the ID countermeasure increases the available leakage in a horizontal manner, either as a fault detection or as a fault tolerance mechanism. Analytically, in the case of an unprotected implementation (without ID) a univariate adversary can acquire the leakage of a key-dependent value $v$, i.e., observe $L_v \sim N(v, \sigma)$, assuming identity leakage model. On the contrary, when instruction n-plication is implemented ($n > 1$), the adversary can observe over time an $n$-dimensional leakage vector $L_v = [L_{v1}, \ldots, L_{vn}]$. The vector contains $n$ independent observations of value $v$ under the same noise level, i.e., we assume that $L_{vt} \sim N(v, \sigma)$, $t = 1, \ldots, n$.

Given that the side-channel adversary has located the sample positions of the repeated leakages, they can perform a pre-processing step where they averages all available samples that leak $v$, i.e., they computes $\bar{L}_v = (1/n) \sum_{t=1}^{n} L_{vt}$. The averaging step results in noise reduction of factor $\sqrt{n}$, obtaining $\bar{L}_v \sim N(v, \sigma/\sqrt{n})$ and as a result side-channel attacks can be enhanced. Note that noise reduction can be particularly hazardous even when additional side-channel protection is implemented. For instance, both masking and shuffling countermeasures [140, 141] amplify the existing noise of a device and will perform poorly if the noise level has been reduced by a large factor $\sqrt{n}$. In order to demonstrate the effect of noise reduction, we employ the information-theoretic framework of Standaert et al. [140] which evaluates the resistance against the worst possible attack scenario. The MI between the key-dependent value $V$ and leakage $L_v$ can be computed using the following formula: $MI(V; L_v) = H[V] + \sum_{v \in V} Pr[v] \cdot \int_{L \in L^v} Pr[v|L] \cdot \log_2 Pr[v|L] \, dL$, where $Pr[v|L] = \sum_{v' \in V} Pr[v|v']$.

From Figure 4.5, we derive the following three conclusions. First, we observe that n-plication (for $n > 1$) shifts the MI-curve to the right, i.e., the FI countermeasure produces repeated leakages which have a direct impact on the side-channel security of...
the implementation. Second, we note that if ID translates to more than two assembly instructions that manipulate the same value, we will likely observe even more hazardous repetitions. Third, it follows that a countermeasure designer needs to balance the need for side-channel resistance and FI resistance by fine-tuning the parameter $n$.

### 4.6.2 Converting infection to ID for SCA

It is important to point out that, apart from straightforward instruction duplication, a wide variety of FI countermeasures rely on some form of spatio-temporal redundancy. For instance, detection methods such as full/partial/encrypt-decrypt duplication & comparison of a cipher \[142] produce repetitions of intermediate values that are exploitable by the side-channel adversary. Thus, an MI-based evaluation of duplication & comparison is identical to Figure 4.5. Similarly, countermeasures that rely on particular error detection/correction codes \[143\] also introduce redundancy that has been evaluated in the side-channel context by Regazzoni et al. \[144\].

In this section, we expand in the same direction and examine the interaction between side-channel analysis and the more recent infective countermeasure \[114\]. Specifically, we demonstrate how the application of a Hidden Markov Model (HMM) \[146\] in a low-noise setting can render infective countermeasures equivalent to ID from a side-channel point-of-view.

Infective countermeasures were developed as a solution to the vulnerabilities of the duplicate & compare methods \[110\]. Instead of vulnerable comparisons, infection diffuses the effect of faults in order to make the ciphertext unexploitable. In particular, we focus on the infective countermeasure of Tupsamudre et al. \[114\], which has been proven secure against DFA \[148\], given that the adversary cannot subvert the control flow and that certain fault models are not applicable \[149\]. The countermeasure is shown in Algorithm 2.

The infective countermeasure alternates between real, redundant and dummy cipher rounds (step 8). It requires an $r$ bit random number $rstr$ (step 3), consisting of $2n$ 1’s that trigger computation rounds (redundant or real) and $(r - 2n)$ 0’s that trigger dummy rounds (steps 5-7). In the event of FI, the difference is detected via function $BLFN : \text{size}(R) \rightarrow 1$, where $BLFN(0) = 0$ and $BLFN(x) = 1$, $\forall x \neq 0$. The error is propagated via step 11.

From a side-channel perspective, the infective countermeasure can be viewed as a random sequence of $r$ round functions, where only the $2n$ computation rounds are useful for exploitation. Thus, the objective of the side-channel adversary is to uncover the hidden sequence of rounds and to isolate the useful ones. Subsequently, one can exploit e.g., the first redundant and first real round together via averaging, which is identical to the afore-mentioned exploitation of ID. Distinguishing effectively dummy rounds from computational ones is non-trivial, especially when extra randomization steps are involved \[150\]. However, the presence of control logic in the infective countermeasure such as variables $\lambda, \zeta$ and $\kappa$ can emit noisy side-channel information about the sequence of rounds. We model such leakage as $L_c = [\Lambda, Z, K] + N(0, \Sigma)$, where the

---

\[145\] Infective countermeasures in this work do not pertain to the modular arithmetic infective techniques used by Rauzy et al. \[145\]
CHAPTER 4. INSTRUCTION DUPLICATION

The suggested HMM is constructed the following way. We encode the main loop of Algorithm 2 using two states, i.e., at a given time $t$, the state $s_t = i \in \{C, D\}$, where $C$ corresponds to a computational round and $D$ to a dummy round. The transitions in the sequence of states is described by matrix $T$, where $T_{i,j} = Pr(s_{t+1} = j | s_t = i)$. Figure 4.4 shows the state diagram and the probabilities for matrix $T$, namely $p = \frac{2n}{r}$. We note that it is possible to unroll the loop and use additional states to describe the transitions, such that we can fine-tune the probabilities. However, we opt for such simple representation to minimize the model’s data complexity.

Algorithm 2: Infection Tupsamudre et al. [114]

Input: Plaintext $P$, key $K$, round $j$ key $k_j$, $\forall j = 1, \ldots, n$, $n$ number of rounds, dummy plaintext $\beta$, dummy round key $k^0$

Output: Ciphertext $C = \text{Cipher}(P, K)$

1: procedure
2: \hspace{1em} $R_0 \leftarrow P$ \hspace{1em} $\triangleright$ Real
3: \hspace{1em} $R_1 \leftarrow P$ \hspace{1em} $\triangleright$ Redundant
4: \hspace{1em} $R_2 \leftarrow \beta$ \hspace{1em} $\triangleright$ Dummy
5: \hspace{1em} $i \leftarrow 1$
6: \hspace{1em} $rstr \in_R \{0, 1\}^r$ \hspace{1em} $\triangleright$ $r$ random bits
7: \hspace{2em} for $q = 1$ until $r$ do
8: \hspace{3em} $\lambda \leftarrow rstr[q]$
9: \hspace{3em} $\kappa \leftarrow (i \land \lambda) \oplus 2(\lnot \lambda)$
10: \hspace{3em} $\zeta \leftarrow \lambda[i/2]$
11: \hspace{3em} $R_k \leftarrow \text{RoundFunction}(R_k, k\zeta)$
12: \hspace{3em} $\gamma = \lambda(-i \land 1)) \cdot \text{BLFN}(R_0 \oplus R_1)$
13: \hspace{3em} $\delta = (-\lambda) \cdot \text{BLFN}(R_2 \oplus \beta)$
14: \hspace{3em} $R_0 \leftarrow (-\gamma \lor \delta) \cdot R_0 \oplus ((\gamma \lor \delta) \cdot R_2)$
15: \hspace{3em} $i \leftarrow i + \lambda$
16: \hspace{3em} $q \leftarrow q + 1$

return $R_0$

In the HMM, the round sequence $s = [s_1, \ldots, s_r]$ is unknown, but the adversary is assisted by leakage observations $[l^{e_1}_c, \ldots, l^{e_r}_c]$. To exploit the observations, the HMM associates every state $i \in \{C, D\}$ with an estimated emission probability function, i.e., emission $e_i(l^e_c) = Pr(l^e_c | s_t = i)$. Having established the HMM for our scenario, we perform a simulated experiment where we try to identify the round sequence for a gradually increasing noise level. The simulated sequence contains 22 computational rounds and 78 dummy rounds, i.e., it corresponds to a computation of AES-128 using infection with $r = 100$. For every noise level we apply the Viterbi algorithm [151], which can recover the most probable sequence $s$ of length $r$, while factoring in the leakage observations $l^{e_1,...,e_r}_c$ and the transition probabilities of $T$. The simulation (Figure 4.6) shows that for fairly small noise levels (e.g., $\sigma < 0.3$) we are able to uncover the hidden sequence with...
4.6. SCA OF ID AND INFECTION COUNTERMEASURES

\[ T = \begin{pmatrix} C & D \\ D & C \end{pmatrix} \begin{pmatrix} p & 1-p \\ p & 1-p \end{pmatrix} \]

\[ T_{pr} = \begin{pmatrix} C & D \\ D & C \end{pmatrix} \begin{pmatrix} p & 1-p \end{pmatrix} \]

Figure 4.4: The Markov model describing the states, transition probabilities \( T \) and prior probabilities \( T_{pr} \).

high probability, making the side-channel exploitation of infection equivalent to the exploitation of instruction duplication.
Figure 4.5: MI of instruction $n$-plication.

Figure 4.6: Success rate of HMM-based sequence detection vs. noise level $\sigma$. 
4.7 Practical SCA results

In this section, we apply the exploitation techniques of Section 4.6.1 in our experimental setup that protects an AES-128 implementation using ID. We verify the technique’s applicability to real-world scenarios by showing their increased efficiency compared to standard SCA methods. We use an AVR MCU (XMEGA128D4) as the main target for our SCA experiments and we collect power traces using the open-source ChipWhisperer product. The clock frequency of the target is 7.3728 MHz and we sample the power consumption of the target 4 times per clock cycle.

We use three different code patterns (shown in Table 4.3) to evaluate the interaction between SCA and ID in different scenarios. Patterns A and B demonstrate how ID affects different instructions, namely instructions eor and ld respectively. Pattern C showcases the duplicated key addition and Sbox parts of a lookup-table-based AES implementation.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Code Snippet</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>eor R10, R17</td>
</tr>
<tr>
<td>B</td>
<td>ld R10, Y</td>
</tr>
<tr>
<td>C</td>
<td>eor R9, R17</td>
</tr>
<tr>
<td></td>
<td>add R28, R9</td>
</tr>
<tr>
<td></td>
<td>ld R10, Y</td>
</tr>
</tbody>
</table>

Table 4.3: AVR code snippets for the SCA experiments. Y is the output buffer and R17 contains the hardcoded secret key.

4.7.1 Horizontal exploitation using CPA

For the afore-mentioned patterns, we perform an experimental evaluation where we put forward a variant of the traditional Correlation Power Analysis (CPA) [152]. In the case of n-plication, we involve a horizontal averaging pre-processing strategy as follows.

1. Locate the intervals pertaining to the n different repeated leakages. In every interval, heuristically select the point in time with the highest correlation to the targeted key-dependent value, obtaining vector \( \mathbf{l} = [l_1, \ldots, l_n] \).

2. For every vector \( \mathbf{l} \) compute the average value \( \bar{l} = (1/n) \sum_{i=1}^{n} l_i \), thus reducing the noise level.

3. Perform CPA using the averaged values (\( \bar{l} \)).

In Figure 4.8, we observe how the averaged CPA using a Hamming weight model outperforms naive CPA that ignores horizontal leakage, since it requires less traces to converge. Thus, the theoretical results of Section 4.6.1 are confirmed in practice and we conclude that horizontal averaging rejects noise. In addition, the difference

https://newae.com/tools/chipwhisperer/
between the naive CPA on the original code and averaged CPA on the duplicated code is larger on the duplicated eor pattern rather than on the duplicated ld. This behavior is attributed to the SNR of ld/st instructions, which is significantly higher compared to the SNR of ALU operations (such as eor) since the later do not manipulate the memory bus. As a result, there is less need to reject noise on memory instructions. Last, we observe that a naive CPA attack when ID is in place may be slower to converge due to interference between duplicated consecutive instructions.

This work focuses on n-plication used as a fault tolerance mechanism, the same averaging technique can be applied when n-plication is used as a fault detection mechanism. In the latter case the instruction stream is the same as in the former case when no faults are injected, therefore, the side channel is similarly amplified.

### 4.7.2 Horizontal exploitation using templates

In order to fully exploit the available horizontal leakage, we also use a template-based approach [153, 154], which comprises two phases for attacking an AES-128 implementation: a profiling phase, in which templates are built for 256 key candidates of an AES-128 key byte and an extraction phase, where a number of traces are used to recover the unknown key. In our experiments, for the profiling phase, we use 3.2k traces of the device per key candidate and perform dimensionality reduction, selecting Points of Interest (POIs) via Principal Component Analysis [155]. We deployed the following two template attacks. To ensure that the side-channel effect of ID is exploited during the heuristic step of POI selection, the first attack breaks the trace in multiple intervals, each containing a single assembly instruction and performs POI selection in every interval separately. The second template attack considers the full

---

\[^{7} \text{SNR}(A)=2.23 \text{ and } \text{SNR}(B)=18.20\]
4.7. PRACTICAL SCA RESULTS

Figure 4.8: Success rate of the CPA attack. *single* is CPA on the original code. On duplicated code, *no-avg* is the naive CPA and *avg* is the CPA with averaging.

trace as a single interval and performs POI selection in the whole region.

In Figure 4.7 we focus on code pattern (C). We perform the CPA attack (naive and averaged) that exploits the duplication of the ld instruction computing the Sbox output. Moreover, we perform the multi-interval and single-interval template attacks. We observe that both template attacks achieve similar performance and surpass the averaged CPA. Thus, we verify the applicability of templates in a horizontal context and conclude that they constitute an optimized way to exploit repeated leakages. We
note that template attacks are inherently multivariate and may often require an extensive profiling phase to effectively characterize the model. On the other hand, averaged CPA compresses multiple samples, i.e., it is a univariate technique with a less informative model compared to templates, yet it has the upside of being faster to train and compute.

4.8 Conclusion

In this chapter we analyzed the limitations of Instruction Duplication (ID) as a fault tolerance mechanism. First, we proved that the model under which ID operates has fundamental limitations, rendering the ID ineffective or even harmful. ID is designed under the assumption of a single fault model. However, in practice a more complex model can hold for a specific target, thus relying only on ID as a fault tolerance mechanism is not effective against FI attacks.

Second, the information leakage through side channel is amplified. We showed that the side channel introduced by instruction by ID, can be successfully exploited to extract secret information. Moreover, other instruction redundancy based defenses suffer from the same weaknesses in respect to side channels.

Finally, while automatically applying redundancy based defenses is promising, the FI model has to be fine tuned and extended for each targeted device according to its runtime configuration. The compiler must use this model to carefully balance fault tolerance guarantees and performance. Whether or not this is possible is still an open question.
Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks

Given the increasing impact of Rowhammer, and the dearth of adequate other hardware defenses, many in the security community have pinned their hopes on error-correcting code (ECC) memory as one of the few practical defenses against Rowhammer attacks. Specifically, the expectation is that the ECC algorithm will correct or detect any bits they manage to flip in memory in real-world settings. However, the extent to which ECC really protects against Rowhammer is an open research question, due to two key challenges. First, the details of the ECC implementations in commodity systems are not known. Second, existing Rowhammer exploitation techniques cannot yield reliable attacks in presence of ECC memory.

In this chapter, we address both challenges and provide concrete evidence of the susceptibility of ECC memory to Rowhammer attacks. To address the first challenge, we describe a novel approach that combines a custom-made hardware probe, Rowhammer bit flips, and a cold boot attack to reverse engineer ECC functions on commodity AMD and Intel processors. To address the second challenge, we present ECCploit, a new Rowhammer attack based on composable, data-controlled bit flips and a novel side channel in the ECC memory controller. We show that, while ECC memory does reduce the attack surface for Rowhammer, ECCploit still allows an attacker to mount reliable Rowhammer attacks against vulnerable ECC memory on a variety of systems and configurations. In addition, we show that, despite the non-trivial constraints imposed by ECC, ECCploit can still be powerful in practice and mimic the behavior of prior Rowhammer exploits.

5.1 Introduction

Originally designed to handle accidental and rare occurrences of data corruption in DRAM chips due to cosmic rays or electrical interference [156][157][158][159], Error-
Correcting Code (ECC) memory is also perceived as one of the few effective bulwarks against Rowhammer attacks \[160\]. These attacks exploit a vulnerability in DRAM hardware that allows attackers to flip bits in memory that should not be accessible to them \[161\]. Since the discovery of the Rowhammer vulnerability in 2014, the security community has devised ever more worrying exploitation techniques. Starting with fairly simple, probabilistic corruption of page tables from native x86 code \[161\], researchers have extended the Rowhammer attack surface across all sorts of computing systems (including PCs \[161\], \[162\], \[163\], clouds \[164\], \[165\], and mobile devices \[166\], \[167\]), launching exploits from different environments (such as native C binaries \[161\] and browser-based JavaScript \[163\], \[162\], \[167\]), using a variety of processors (notably x86 \[161\], ARM \[166\], and GPU \[167\]), against a variety of targets (page tables \[161\], \[166\], encryption keys \[165\], object pointers \[162\], repository URLs \[165\], and opcodes \[168\]), in different types of memory (DDR3 \[161\] and DDR4 \[166\]). As a result, Rowhammer has grown into a major security concern in real-world settings.

Not surprisingly, there has been much speculation on the effectiveness of ECC memory in deterring real-world Rowhammer attacks \[160\], \[165\], \[166\], \[161\], \[168\], often hypothesizing ECC memory would reduce Rowhammer to a denial-of-service vulnerability \[161\], \[168\]. As a result, practical Rowhammer exploits have thus far only targeted non-ECC-equipped platforms. However, once the uncommon case, ECC-equipped platforms are now on the rise, from large cloud providers (e.g., Amazon EC2 \[169\]) to high-end consumer platforms \[170\]. In addition, ECC memory is increasingly deployed on low-power platforms such as mobile and IoT devices to drop the DRAM refresh rate below “safe” values and save power \[171\], \[172\]. It has therefore become important to quantitatively assess the effectiveness of ECC memory as a Rowhammer mitigation.

ECC is able to correct \(n\) bit errors (with \(n \geq 1\)) and detect cases where more than \(n\) bits have flipped, up to some maximum. For this purpose, ECC adds redundant ECC bits to every data word that “check” the other bits. The combination of the data bits and the ECC bits is known as a code word. ECC ensures that if any bit in a valid code word changes, it is no longer a valid code word. Thus, in a chipset with ECC memory, attackers may still use Rowhammer to cause a bit flip in physical memory, but the ECC mechanism immediately catches it on the first subsequent access, and flips it back. Since the probability of flipping exactly the right set of bits to turn one valid code word into a new valid code word using Rowhammer is extremely low, state-of-the-art Rowhammer attacks either fail, or trigger uncorrectable errors, leading to denial of service. Better still, modern processors apply additional memory reliability measures such as data masking (scrambling) to turn the data that the CPU really writes to main memory into pseudo-random patterns—making it even harder for an attacker to flip the right bits. The research question in this chapter is whether the assumption is true that Rowhammer attacks are really not practical on ECC memory. In particular, we examine the strength of ECC in several modern chipsets and show that this is not the case: reliable attacks in real-world settings are harder, but still possible.

To determine the exact protection offered by ECC, we must know the details of the ECC algorithms. Unfortunately, vendors such as Intel and AMD do not release these details. Moreover, to the best of our knowledge, no prior work has managed to reverse engineer the ECC functions. Important contributions of this chapter are therefore the
recovered ECC computation for popular chipsets and a detailed description of the *techniques* to reverse engineer other ECC algorithms.

A major challenge in examining a DRAM’s susceptibility to Rowhammer on ECC memory, both for us and for attackers, is detecting the bit flips in the first place. How do we even know that we flipped a bit using Rowhammer, if the hardware automatically flips it back when we try to read it? Phrased differently, *observing* ECC errors is hard, precisely because the hardware is designed to hide them. To solve this problem, we describe a novel side channel that allows us to observe bit flips even when the error correction functionality flips them back when we read the corresponding memory location.

Armed with the ability to detect (correctable) bit flips and knowledge of a fully reverse engineered ECC algorithm, another challenge towards reliable attacks is to surgically trigger the “right” combination of bit flips in a single code word to bypass ECC. An invalid combination may be corrected or, worse, trigger uncorrectable errors and crash the system. To address this challenge, we develop a new Rowhammer attack technique based on composable, data-controlled bit flips. The key insight is that Rowhammer bit flips are data-dependent and, if we study how specific data patterns determine the triggering of individual bit flips, we can then reliably isolate/compose multiple bit flips by placing the “right” data patterns in memory. Our attack, termed *ECCploit*, relies on such insight to incrementally find an exploitable combination of bit flips in a code word and bypass ECC memory.

Given the need to bypass ECC checks, such exploits are more constrained compared to existing Rowhammer attacks. For this reason, we reproduce known end-to-end exploits on ECC memory and analyze the attack surface, that is the probability of finding the bit flip patterns that bypass the ECC checks for these exploits. While we do find that ECC checks significantly reduce the Rowhammer attack surface, we show *ECCploit* can still be used to successfully mount Rowhammer exploits in practical settings. In addition, while we evaluate *ECCploit* in an ideal scenario where the system is configured properly to handle ECC errors (i.e., the worst case for attackers), we find that in many systems this is not the case. For example, while we expect a crash in case of uncorrectable errors, sometimes the system does not immediately crash, allowing for much simpler exploitation with ECC memory.

**Contributions.** Our main contribution is showing that ECC memory, even when combined with data scrambling, does not offer adequate protection against Rowhammer. We do so by:

- Describing a novel reverse engineering technique for recovering ECC implementations on commodity hardware.
- Identifying the ECC implementation on several popular chipsets and investigating how commodity systems respond to ECC exceptions.
- Presenting *ECCploit*, a new reliable Rowhammer attack that leverages undocumented ECC implementation details, a novel side channel in the memory controller, and composable, data-controlled bit flips. We show *ECCploit* can be used for practical privilege escalation attacks by reproducing existing exploits on ECC-based systems.
5.2 Background

In the following, we provide a high-level description of the DRAM architecture, the Rowhammer vulnerability, and ECC properties we rely on for our *ECCploit* attack.

5.2.1 DRAM organization

**Architecture.** DRAM uses one of the last parallel buses in modern systems. In a common setup, 64 lines connect a Dual Inline Memory Module (DIMM) to the CPU forming a 64-bit wide data bus. Multiple chips inside a DIMM form the 64 bits of data every time DRAM is accessed. For example, with 8-bit wide chips (i.e., 8x), eight chips are involved in each DRAM read or write operation. Each chip consists of multiple banks. Multiple rows of DRAM cells are stacked together to form each of these banks. Cells are the smallest unit of storage in DRAM and are built using a capacitor and an access transistor. The amount of charge stored in the capacitors denotes the value of one or zero depending on the charge level.

**Accessing DRAM.** The smallest unit of access inside DRAM is a row. To access DRAM, the same bank is selected in all chips and the data from the selected row is moved to a cache called *row buffer* before being transmitted on the bus (i.e., row activation). Subsequent accesses to addresses that map to the same row will be served from the row buffer (i.e., row hit) and addresses that map to a different row require writing the contents of the row buffer back to the cells and moving the target row into the row buffer (i.e., row miss).

**Refresh.** Given that DRAM cells are built from capacitors, they lose charge and hence their value over time. To restore the charge, the cells need to be recharged, a process called *DRAM refreshing*. This process is orchestrated by the memory controller, which is responsible for periodically refreshing individual DRAM cells at a predetermined *refresh rate*. The refresh rate is determined based on the expected amount of charge leakage (e.g., dependent on the manufacturing process), and the implementation constraints (e.g., presence of ECC).

**Supporting ECC.** Cosmic rays and other external events can cause corruption in DRAM cells by changing the charge levels in the capacitors [158, 159, 157, 156]. To address this problem, ECC memory stores extra parity bits (also known as control bits) next to the data bits to correct these corruptions. DRAMs with ECC support come with additional chips. The memory bus is then enlarged with eight additional lines (i.e., 72-bit wide bus) to transfer the control bits next to their data bits [173, 174, 175, 176].

5.2.2 Rowhammer

As transistors become smaller, their reliability starts to suffer. Kim et al. [160] showed that frequent activations of the same row cause bits to flip in adjacent rows without accessing them. The reason is the increased amount of charge leakage from DRAM cell capacitors (built from transistors) due to parasitic coupling and passing gate effects. Termed the Rowhammer vulnerability, soon a plethora of attacks abused a single bit
flip to compromise desktops, laptops, and mobile phones [166, 165, 161, 162, 163, 167, 168]. Such attacks come in different variants, double-sided, single-sided, or one-location Rowhammer [168]—depending on the aggressor row(s) used by the attacker to corrupt the victim row—and exploit the fact that Rowhammer bit flips are observable and reproducible.

All of these attacks have been executed on systems without ECC and, while there has been speculation on the possibility of bypassing simple ECC functions since the original Rowhammer paper [160], an end-to-end Rowhammer attack on ECC memory on a real system has never been attempted for two main reasons. First, ECC implementations on modern systems are often undocumented and go beyond the simple SECDED ECC which we describe shortly. Second, it is challenging to trigger Rowhammer corruptions without triggering corrections or crashes on a system protected by ECC. Before further discussing these challenges, we need to understand how ECC is currently implemented on modern commodity systems.

5.2.3 ECC in DRAM

In current designs, the only ECC-aware unit inside the processor is its memory controller. Assuming the CPU wants to write a message of \( k \) bits, the memory controller appends \( r \) bits of redundant information for error correction and detection and stores a codeword of \( n = k + r \) bits in DRAM. In practice, CPU vendors choose \( k \) to be a multiple of a memory word (64 bits) and \( r = \frac{k}{8} \). In fact, the ratio of redundant to data bits (1-to-8) is embedded in the current Double Data Rate (DDR) standards (DDR3 [173] and DDR4 [174]), memory bus standards with 8 control bits and 64 data bits. For manufacturing simplicity, the same type of memory chips is used to store both the data bits and the control bits. Concretely, one can identify DIMMs that provide ECC by counting the number of memory chips on the module.

**Block codes.** DRAM ECC uses linear block codes for calculating the \( r \) bits [177]. Differences in the size of \( r \) bits and their actual value provide different trade-offs in terms of reliability and performance. There are two types of linear block error correcting codes, binary and non-binary codes. A binary code is denoted as \((n, k)\) and has a granularity of a single bit while non-binary codes treat multiple bits as a single symbol. A particular case of binary code, the \((7, 4)\) code, was first studied and generalized by Richard Hamming [178] and represents an improvement from the simple parity checking as it offers error correcting capabilities with 3 parity bits for 4 bits of data.

**SECDED.** The Hamming Distance (HD) between any codeword \((d_{\text{min}})\) of the \((7, 4)\) code is at least 3, meaning that it can detect up to 2 bit errors and correct a single detectable error. However, distinguishing between a message that has a corruption of one bit and a message that has a corruption of two bits is not possible. The implication is that some 2-bit faulty messages will falsely be “corrected”. An extended Hamming code adds an extra parity bit to solve this problem and serves as the basis of the design of ECC used in modern memory systems as it provides single error correction and double-bit error detection (SECDED) [177].

**Chipkill.** High-available systems need to detect multiple adjacent bit errors. This
requirement of the error correcting capabilities is known as the chipkill functionality. BCH codes have the desired property of precise control of the error guarantees. The Reed–Solomon (RS) codes are a class of effective and easy-to-construct non-binary codes which can be viewed as particular BCH codes. The commonly deployed Chipkill implementation, based on BCH/RS codes, provides double-chip error detect and single-chip error correct (SCDCD). Note that Chipkill can correct bit errors up to the size of the symbol, which is often chosen to be the number of bits in a chip. As a result, even if the system loses an entire chip, it can still continue operation.

More generally, a linear block error detecting and correcting code with a $d_{\text{min}}$, can detect $d_{\text{min}} - 1$ errors and correct $\lfloor (d_{\text{min}} - 1)/2 \rfloor$ errors. Similarly, an RS code that can correct $t$ symbols has a HD of $2t + 1$ and uses $2t$ redundant error correcting symbols.

As we shall see in Section 5.5.6, our setups use a version of RS codes.

### ECC functions.

For simplicity and compatibility with non-ECC DIMMs, it is desirable for the memory controller to store the control bits and the data in distinct memory chips. From a theoretical perspective, this requirement maps over the systematic encoding procedure, in which the message is always a prefix in the codeword.

To encode a message $d = (d_1, d_2, \ldots, d_k)$, where $d_i$ represents a symbol from the alphabet (e.g., a bit), the encoder performs a multiplication with a generator matrix $G$, i.e., $v = d \cdot G$, where $v$ is the encoded message (data). For the practical systematic encoding procedure, $G = [I_k | P]$, where $I_k$ is the identity matrix of size $k$, and $P$ is the parity check matrix which has $k$ rows and $r$ columns:

$$v = d \cdot G$$
$$= d \cdot [I_k | P]$$
$$= d \cdot ([I_k[0_{k,r}] + [0_{k,k}|P])$$
$$= d \cdot [I_k[0_{k,r}] + d \cdot [0_{k,k}|P]$$  \hspace{1cm} (5.1)

Let $\text{ECC}(d)$ be the last $r$ bits from the $d \cdot [0_{k,k}|P]$ product, which we loosely call the ECC bits for data $d$. Using the Kronecker function ($\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ if $i \neq j$), we can rewrite the ECC bits as:

$$\text{ECC}(d) = \sum_{i=1}^{k} d_i \cdot [\delta_{1,i} \delta_{2,i}, \ldots, \delta_{k,i}]$$
$$\text{ECC}(d) = \sum_{i=1}^{k} d_i \cdot [P_{i,1}, P_{i,2}, \ldots, P_{i,r}]$$  \hspace{1cm} (5.2)

where $P_{i,j}$ represents the value (0 or 1) from the parity check matrix with coordinates row $i$ and column $j$. Each row of the parity check matrix can be expressed as an $r$ bit number called parity value. Parity check matrices are not disclosed by processor manufacturers. We devise techniques for obtaining this information on various systems in Section 5.5. Once we have the parity check matrix, we can predict ECC values for arbitrary data. On top of ECC, some systems further scramble data before sending them on the memory bus, complicating the reverse engineering of parity check matrices.

### 5.3 Threat model

We assume computer systems protected with ECC memory where bit flips are detected and/or corrected in the memory controller. This is common in clouds, high-end
workstations, and low-power devices. We further assume the memory chips to be affected by the Rowhammer vulnerability \[160\]. In addition, we assume that the attacker does not have access to ECC exceptions as these are often exposed to privileged software. Thus the attack can be carried by a non-privileged local user. We assume that the attacker can learn the CPU model and the memory technology. This is trivial to satisfy as access to /proc/cpuinfo is unrestricted and cloud providers’ public documentation usually contains a description of the underlying hardware \[182\] \[183\]. Similar to existing Rowhammer attacks, the attackers’ aim is to reliably compromise co-located virtual machines \[184\] \[165\] or escalate their privilege by executing unprivileged and/or sandboxed code on the target machine \[166\] \[161\] \[162\] \[163\] \[167\] \[168\].

## 5.4 Summary of challenges

To exploit a system protected with ECC memory using Rowhammer, the attacker first needs to find the ECC algorithm implemented in the memory controller of the target system’s processor. Given the knowledge of the ECC function, the attacker then needs to safely compose enough bit flips to trigger a Rowhammer corruption that is not detected (and corrected) by the ECC algorithm—without triggering uncorrectable errors that may crash the system. These corruptions are different than normal Rowhammer corruptions given that they flip multiple bits at the same time. Because the probability of bits to be in the “flips-from” state decreases as the number of bits that flip increases, it becomes challenging to exploit such constrained bit flips to compromise a system. In summary, to achieve successful and reliable end-to-end exploitation, we need to address the following challenges:

[C\(_1\)] How to reverse engineer unknown ECC functions on commodity processors?

[C\(_2\)] How to trigger Rowhammer corruptions on ECC memory without crashing the system?

[C\(_3\)] How to exploit the system given that Rowhammer-based ECC corruptions corrupt multiple bits at the same time?

We address [C\(_1\)] in Section 5.5, [C\(_2\)] in Section 5.6, and [C\(_3\)] in Section 5.6.2 and in Section 5.7.

<table>
<thead>
<tr>
<th>ID</th>
<th>Manufacturer</th>
<th>CPU model</th>
<th>Microarchitecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD-1</td>
<td>AMD</td>
<td>Opteron 6376</td>
<td>Bulldozer (15h)</td>
</tr>
<tr>
<td>Intel-1</td>
<td>Intel</td>
<td>Xeon E3-1270 v3</td>
<td>Haswell</td>
</tr>
<tr>
<td>Intel-2</td>
<td>Intel</td>
<td>Xeon E5-2650 v1</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>Intel-3</td>
<td>Intel</td>
<td>Xeon E5-2620 v1</td>
<td>Sandy Bridge</td>
</tr>
</tbody>
</table>
5.5 Challenge $C_1$: reverse engineering ECC

To get a rough idea of the ECC functions used by CPU manufacturers, we first consulted their patents and the CPUs’ public documentation. Unfortunately, these were neither complete nor fully accurate, so additional techniques were necessary. As we shall see, the coding theory behind our attacks is quite involved, so we first provide the intuition.

Whenever an ECC system writes a value in memory, it will also write some ECC bits. For instance, some ChipKill implementations write 4 ECC nibbles (for a total of 16 bits) for every 128 bits of data. The exact calculation of the ECC nibbles is not important at this point, but the first ECC nibble will use one set of data nibbles, the second one a slightly different set, and so on. Upon accessing this value in memory at a later stage, it will calculate the ECC nibbles again and XOR them with the ECC nibbles in memory. The result is known as a syndrome. If the syndrome is non zero, there must have been an error. By looking at which syndromes indicate an error, ChipKill can locate the faulty nibble and correct it.

As we shall see, the calculation of the syndromes in mathematical terms involves a fairly complicated multiplication of the transposed and extended parity check matrix with the error pattern, but in practice the multiplication matrix is precomputed and stored as a table, while the multiplications and additions are simply AND and XOR operations (as shown above). The point is that if we have the syndromes for known error patterns, we can also perform the inverse operation and obtain the parity check matrix—and hence the ECC function.

To this end, we artificially injected single bit errors in memory to see what happens and deduce what the syndrome must have been, and also performed cold boot attacks to recover the ECC bits as generated by one machine on another machine. We detail these techniques after providing a theoretical foundation for the attacks. To our knowledge, we are the first to reverse engineer the ECC functions of common CPUs (Table 5.1).

5.5.1 Theoretical foundation

Both Hamming and BCH codes are polynomial codes. Polynomial codes can use exclusive-or instead of addition and and instead of multiplication in the Galois Field (GF), simplifying their implementation in hardware.

**Proposition 1.** We can recover the complete ECC function by finding the ECC value for every ECC-word with exactly one data bit asserted.

Each row of the parity check matrix, can be expressed as a $r$ bit number called parity value. Considering Equation 5.2, the ECC value for a data word $(d)$ that has bits asserted on positions $s$, can be expressed as an exclusive-or operation between the parity value of each data word $(d')$ with a single $d'_i$ asserted ($\forall i \in \{s | d_s = 1\}$).

To decode and correct errors of a received codeword $v' = (v_1, v_2, \ldots, v_{k+r})$, linear codes use an efficient technique called syndrome decoding. The syndrome is computed as $S(v') = v' \cdot H^T$ where $H = [-P^T I_{k+r}]$ for the systematic encoding and $S$ has
5.5. CHALLENGE \( C_1 \): REVERSE ENGINEERING ECC

dimensions \((1, r)\). When no error occur in the transmission \((v' = v)\) then \(S(v') = d \cdot [I_k|P] \cdot [-P^T[I_{k+r}]]^T \Rightarrow S(v') = 0.\)

**Proposition 2.** The ECC value of a data word with a single bit asserted on a specific position is equal to the syndrome obtained when that specific bit is faulted.

In the presence of an error \(e = (e_1, e_2, \cdots, e_{k+r})\) with \(e \neq 0_{1,k+r}\), \(v' = v + e\), and because \(S(v) = 0\), we can rewrite the syndrome as:

\[
S(v + e) = (v + e) \cdot H^T = v \cdot H^T + e \cdot H^T = S(v) + e \cdot H^T = e \cdot ([-P^T[I_r]]^T + [0^T[I_r]^T])
\]

We use the notation \(SYND(v') = (e_1, e_2, \cdots, e_k)\cdot\bar{P}\), to refer to the syndrome obtained when errors are inserted only in the data bits. Using the Kronecker function we can rewrite the syndrome obtained under faults as:

\[
SYND(v') = -\left(\sum_{i=1}^{k} e_i \cdot [\delta_{1,i}, \delta_{2,i}, \cdots, \delta_{k,i}]\right) \cdot P
\]

As the operations are performed on a binary GF and the code is cyclic, the “\(-\)” sign has no meaning. Therefore by choosing \(e_i = d_i\) in Equation 5.2 and 5.4 we obtain the proof below. For simplicity, we choose \(e_i\) such that at most one bit is flipped.

\[
\forall v: ECC(v) = SYND(v). \blacksquare
\]

Assuming the attacker has access to the same machine as the victim, we show how an attacker can use Proposition 1 and 2 to inject faults and perform cold boot attacks to reverse engineer the contents of the parity matrix and the order in which the output data is mapped to the DRAM bus lines. Note that the attacker needs to perform this process only once and reuse the recovered information when attacking victim machines that use the same CPU model. The CPU model information on the victim machine is available through sources such as `cpuid`.

5.5.2 Fault Injection

In this section, we describe how to obtain all syndromes (and thus the ECC function) by observing only the syndromes for specific errors that we inject ourselves in a controlled way, where exactly one bit is flipped. For now, we assume that when the ECC engine corrects an error, the attacker can also read the syndrome for that specific error. We will show how we relax this assumption later. The crux of our attack is that if we repeatedly flip a single bit at every possible bit position of an ECC word, and obtain all the corresponding syndromes, the recovery of the ECC function is trivial (Equation 5.5). For example, the ECC value of an ECC word where bit \(i\) and \(j\) are asserted is the result of the XOR operation between the syndrome when a 1-to-0 bit is flipped.
in the $i$ position and the syndrome when the bit is flipped in the $j$ position. To recover the syndromes, we flip bits at the desired bit positions using one of the following three fault injection mechanisms: 1. a custom built shunt probe. 2. facilities provided by some memory controllers. 3. Rowhammer bit flips. We describe these mechanisms next.

**Error injection with a shunt probe.** To reduce noise and cross-talk between high-speed signals, data pins of the DDR DIMM ($DQ_x$) are physically placed next to a ground ($V_{SS}$) signal. As the ground plane ($V_{SS}$) has a very low impedance compared to the data signal and because the signal driver is (pseudo) open drain, short-circuiting the $V_{SS}$ and $DQ_x$ signals will pull $DQ_x$ from its high voltage level to “0”. Depending on the encoding of the high voltage, this short-circuiting results in a 1-to-0 or 0-to-1 bit flip on a given $DQ_x$ line.

Figure 5.1 displays the locations of the important signals and shows that a $DQ_x$ signal is always adjacent to a $V_{SS}$ signal. Therefore, to inject a single correctable bit error, while the system exercises the memory by writing and reading all ones, we have to short-circuit a $DQ_x$ signal with $V_{SS}$. We can achieve the short-circuiting effect with the help of a custom-built shunt probe using syringe needles (Figure 5.2a). We insert the probe in the holes of the DIMM socket as shown in Figure 5.2b. For clarity, we omit the memory module from the picture. We then use tweezers to control when the error is injected by shorts-circuiting the two needles and thus the targeted $DQ_x$ and nearby $V_{SS}$ signal. This method, while simple (and cheap), is effective in the case of a memory controller that computes ECCs in a single memory transaction (ECC word size is 64 bits) and can be used instead of expensive ad-hoc equipment [185, 186].

On some systems (e.g., configuration AMD-1) data is retrieved in two memory transactions and then interleaved. Because of the low temporal accuracy of the shunt probe method, an error inserted on memory line $DQ_k$ ($0 \leq k < 64$) that appears on data bit $2 \times k$ will also “reflect” on data bit $2 \times k + 1$ inside the 128 bit ECC word. In this case the syndrome corresponds to two bit errors and contradicts Proposition [1]. To ensure single bit errors, once the interleaved mechanism is understood, the exercising data can be constructed such that the reflected positions contain only bits that are encoded to low voltage, essentially masking the reflections.

**Error injection with memory controller.** Some server-grade processors incorporate memory controllers that provide the functionality for artificially injecting errors in memory. This mechanism is useful when testing the error-reporting functionality of the software stack. The error injection facility is exposed as PCI registers, but the OEM can choose to lock these resources from the firmware. Furthermore, the way to specify where the error and what type of error is injected varies across platforms. For example, on some systems the error is injected on the next uncached memory access (e.g., AMD-1) while on others the error is injected on an address that is explicitly specified (e.g., Intel-1).
5.5. CHALLENGE $C_1$: REVERSE ENGINEERING ECC

(a) A custom shunt probe.  
(b) Tweezers short-circuiting $DQ_0$ and $V_{SS}$.

Figure 5.2: Fault injection with the help of syringe needles.

**Error injection with Rowhammer.** It is also possible to use Rowhammer to trigger bit flips when support for error injection in the memory controller is lacking. Note that this Rowhammer “attack” is merely intended to detect the syndromes and not (yet) to bypass ECC. When a vulnerable aggressor-victim row is detected (either by observing ECC error counters or by using the side-channel introduced in Section 5.6.1), the position of the bit flip is still unknown to the attacker. However, as we show in Section 5.6.2 we can overwrite the value of the vulnerable bit with the value to which it flips, to stop the bit from flipping under Rowhammer. Therefore, no error is observed when the bit is masked. We can then leverage this property to perform a binary search for the position of the bit flip. The main problem with this method is the need to find bit flips on every possible position within ECC-word size. On the other hand, once attackers own a set of such vulnerable DIMM(s), they can use these DIMMs to reverse engineer any target.

5.5.3 Dealing with lack of syndromes

On some systems, the entire error-handling stack is exposed to software and drivers adequately report the syndromes when ECC errors happen. On other systems, drivers do not always properly report the syndromes (e.g., Intel-1) and on yet other systems, syndromes are lacking altogether (e.g., Intel-2 and Intel-3). We developed our own driver for reading syndromes for Intel-1. For Intel-2 and Intel-3, it is possible to use the available error counters (for which we also developed drivers) and rely on Proposition 1 to reverse engineer the ECC function. However, this approach is error-prone and requires more manual effort. Instead, we rely on a cold boot attack for reverse engineering the ECC functions on these systems.
5.5.4 Cold boot attacks

Cold boot attacks, previously used to breach privacy and reverse engineer the data scrambling performed inside memory controllers [187, 188], consist of three main steps: 1. interesting data is written in memory, 2. the temperature of the memory is lowered such that data retention of the DDR module is high, and 3. the memory is read back after a reboot, for instance by removing the DIMM and immediately plugging it into another machine and booting.

To read the ECC bits, the attacker can perform a cold boot attack, where the first two steps are similar to other cold boot attacks. However, because the ECC bits are not exposed explicitly by the memory controller, we cannot directly access them in Step 3. We can use a custom FPGA-based memory controller to read the ECC control bits. While there are existing solutions to do so for normal DIMMs [189], we did not find a cost-effective solution for ECC memory. Instead, we opted for using an off-the-shelf motherboard and CPU combination for which we already recovered and verified the ECC function with methods presented in Section 5.5.2. Knowing 1) the data that was written, 2) the data that we read after the cold boot, 3) the expected ECC value and 4) the observed syndrome, we can reconstruct the ECC value that was stored by the victim system for certain data patterns.

One challenge is that ECC memory is normally always initialized at boot time by the target system to avoid spurious ECC errors when accessing the memory. This initialization is usually done by the firmware (BIOS) and stops us from performing our cold boot attack. To achieve our goal, we bypassed the memory initialization by reverse engineering and modifying the parts of the binary BIOS code that performs DRAM initialization. We will open-source this patch along with all other necessary details to allow others to build a generic ECC memory dumper.

5.5.5 Reverse engineering approach

Table 5.2 summarizes the pros and cons of our available reverse engineering mechanisms. We now briefly describe how we employed these mechanisms to reverse engineer ECC functions on the machines described in Table 5.1.

Machine AMD-1. Here, the data sheet includes the syndrome table decoding technique for locating ECC errors. The system supports symbols of 4 or 8 bits wide and uses 128 bits (two 64-bits beats interleaved) to compute the ECC control bits. The data sheet further claims that the code can correct any number of errors in a single symbol and detect two symbols data corruption, hinting at a variant of the BCH code. We recover the complete ECC function using the syndrome table. To find out that the system indeed uses the same ECC functions to find the mapping of the data bits to DRAM pins, we employ our shunt probe. Our results conclude that AMD-1’s memory controller accurately reports errors and we further find how data bits are mapped to DRAM pins. The mapping of data bits to DRAM pins is helpful when reverse engineering with cold boot attacks.

The data sheet of a newer version of the AMD-1 CPU model mentions the support for error injection. We therefore wrote a driver for injecting errors through the memory controller of this system and confirmed that it also supports this mechanism. We used
the error injection functionality to also confirm that bit errors in different symbols are uncorrectable.

**Machine Intel-1.** The ECC function for this system is not documented. While it has support for error injection through the memory controller, unfortunately driver support for this functionality at the moment of writing is non-existent. Given that writing a device driver for error injection in this processor is much more involved than just reading information (such as syndromes), we opted for using Rowhammer bit flips themselves for reverse engineering the ECC function. The data sheet of Intel-1 exposes the ECC error counters and syndromes of the ECC error. We had to write our own drivers to access this information. We previously already built a database of vulnerable bits and DIMMs and used a novel side-channel attack to leak whether the ECC unit is correcting a bit flip (which we explain in Section 5.6.1). Using our database of bit flips on these vulnerable DIMMs, we found the syndromes for each vulnerable bit position—only three DIMMs were required for a complete recovery. We validated our results using the shunt probe, which showed that the memory controller shuffles the data when sending them to various data pins on the DIMMs.

**Machines Intel-2 and Intel-3.** These two machines are the least friendly in terms of documentation, but their data sheets do mention that ECC is generated over 64 bits of information at a time. Using our shunt probe, we realized that the software stack in these machines does not report ECC errors. To reverse engineer the ECC functions on these machines, we employ our cold boot attack and rely on the already reverse engineered ECC function on AMD-1 to stage the last step of the cold boot attack. We re-flashed the BIOS of AMD-1 with changes that bypass the memory initialization. In this process, we used an old version of the memory initialization that was contributed by the manufacturer to the coreboot project [190]. Note that the two-beats ECC computation and residual errors due to cold boot complicate the complete recovery of the parity matrix on these machines. As a result, the recovered ECC functions for these machines still contain a few incorrect cases.

### 5.5.6 Results

For brevity, since AMD-1 and Intel-1 are representative of the general trends we observed across all setups, and the recovery on Intel-2 and Intel-3 is not entirely complete due to residual errors in the cold boot attacks, we focus on AMD-1 and Intel-1 in the remainder of the chapter. Even so, all the recovered parity matrices for the configurations in Table 5.1 can be found in Figures 5.4 and 5.5. Furthermore, to quickly visualize the error correcting capabilities, in Figure 5.3 we show the HD of the recovered ECC algorithms. A pixel of coordinate \(x, y\) has a brightness level of the HD between the ECC result of \(data_x\) and \(data_y\). Where \(data_i\) means that bit on position \(i\) is asserted and all the others are de-asserted. A black pixel (lowest brightness and HD) means that the ECC are the same. On AMD-1 (Figure 5.3a) we observe a distinct pattern at 8 bits intervals. This is expected, as the ECC algorithm treats 8 bits as a single symbol. Repetitions are also observed in Figure 5.3d at 4 bits. This implementation corresponds to an Intel patent [191] which can detect up to 4 bits (SEC-DED-S4ED). These patterns are not always obvious, for example Intel-1 (Figure 5.3c) uses the same ECC algorithm (and values) but the bits are shuffled.
Ideal guarantees. We now discuss the ideal guarantees provided by the ECC functions in the two representative systems (AMD-1 and Intel-1). In an ideal setting, correctable errors should be detected and corrected, while uncorrectable errors that are detected should result in a process or system crash. In this configuration, the only way an attacker can compromise the system is by triggering enough bit flips at the right positions to ensure that the ECC function does not detect a corruption. Table 5.3 shows the minimum number of bit flips required in either data bits (i.e., $d_{\text{min}}(\text{data})$) or data bits plus control bits (i.e., $d_{\text{min}}(\text{cw})$). Triggering these many bit flips close to each other is difficult on most DIMMs that are vulnerable to Rowhammer. However, it is much easier to trigger corruptions on Intel-1 as discussed next.

State of practice. As shown in Table 5.4, we found that in Intel-1 detected uncor-
5.5. CHALLENGE $C_1$: REVERSE ENGINEERING ECC

Rectable errors do not crash the system and are not even reported by the OS. The main cause seems to be improper software support for the memory controller in the OS, i.e., the error reporting driver fails to recognize and initialize the resources of the error.

<table>
<thead>
<tr>
<th>ECCIntelHaswell</th>
<th>ECCAndFam10h</th>
</tr>
</thead>
<tbody>
<tr>
<td>111000101</td>
<td>0101011011000</td>
</tr>
<tr>
<td>011000011</td>
<td>010000001001</td>
</tr>
<tr>
<td>001100011</td>
<td>001010000001</td>
</tr>
<tr>
<td>000101101</td>
<td>000010001010</td>
</tr>
<tr>
<td>100110000</td>
<td>011000001001</td>
</tr>
<tr>
<td>000011011</td>
<td>010010000011</td>
</tr>
<tr>
<td>100010010</td>
<td>001100000110</td>
</tr>
<tr>
<td>110100000</td>
<td>100011000001</td>
</tr>
</tbody>
</table>

Figure 5.4: Recovered parity matrices (Intel-1 and AMD-1 respectively).
reporting mechanism. As a consequence, an attacker can exploit the system, in its default configuration, with a smaller number of bit flips than necessary with the ideal guarantee provided with the ECC function.

Exploitable patterns. We use Z3, a constraint solver, to mine exploitable patterns of the ECC functions for AMD-1 and Intel-1. Table 5.5 shows the results for the ideal and default configurations. For AMD-1, the attacker requires at least three bit flips in 16 bytes (i.e., an ECC word) when one of the bit flips is in the control bits ($P_1$). The other two bit flips should target two distinct symbols (i.e., be at least 8 bits apart). When targeting data bits alone, four bit flips should land in at least two distinct symbols in an ECC word ($P_2$).

For Intel-1, in an ideal configuration, an attacker needs to find four bit flips in at least two distinct symbols (i.e., at least 4 bits apart) in eight bytes ($P_3$). However, given that Intel-1 does not crash on detected uncorrectable errors, with only two bit flips in distinct symbols in an ECC word, it is possible to exploit the system ($P_4$).

Exploitable ECC DIMMs. We ordered ECC DIMMs from four different DRAM

Figure 5.5: Recovered parity matrix with cold boot attack on Intel-2 and Intel-3.
5.5. CHALLENGE $C_1$: REVERSE ENGINEERING ECC

chip manufacturers. We chose ECC DIMMs with DRAM chips based on previously published work [165, 192, 193]. Note that the exact same DRAM chips are used both in ECC and non-ECC DIMMs. We found that one out of the four manufacturers produces DIMMs that cause corruption on both AMD-1 and Intel-1. Table 5.6 shows the results of hammering 109k pairs of aggressor-victim-rows and the percentage of rows that have enough bit flips to escape the patterns discussed in Table 5.5. We later use this DIMM to evaluate our end-to-end exploits in Section 5.7.

Other DIMMs. Table 5.7 shows the ECC protection for the public database of bit flips published by Tatar et al. [193] that contains 14 desktop DIMMs with the kind of chips that are used in ECC DIMMs also. We find that every DIMM but one exhibits bit flips that ECC cannot correct and 10 contain potentially uncorrectable corruptions that the ECC algorithm cannot detect. When the ECC detection is used correctly (i.e., $P_1$, $P_2$ and $P_3$), 0.65%-7.42% of all bit flips still cause silent corruptions. On the default configuration ($P_4$), on average up to 10.89% of the bit flips cannot be corrected.
Table 5.2: Advantages (△) and disadvantages (▽) of the proposed ECC recovery methods in this chapter (⊖ indicates ‘neutral or fixable’).

<table>
<thead>
<tr>
<th>Method</th>
<th>Compatibility</th>
<th>Price</th>
<th>Setup Time</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needle FI</td>
<td>△△ works on any hardware</td>
<td>△△ a few dollars</td>
<td>▽ fiddly</td>
<td>△△ recovers signal mapping</td>
</tr>
<tr>
<td>Mem. cntr.</td>
<td>▽▽ not always available</td>
<td>△△ free</td>
<td>▽ software support is rare</td>
<td>⊖ potentially imprecise no signal mapping</td>
</tr>
<tr>
<td>Rowhammer FI</td>
<td>△ targets’ performance</td>
<td>△ vulnerable DIMMs</td>
<td>△ quick</td>
<td>▽ no signal mapping</td>
</tr>
<tr>
<td>Cold boot</td>
<td>△△ works on any hardware</td>
<td>▽ initial investment</td>
<td>▽ rather slow</td>
<td>△△ recovers signal mapping</td>
</tr>
</tbody>
</table>

- △ indicates an advantage.
- ▽ indicates a disadvantage.
- ⊖ indicates a neutral or fixable feature.
Table 5.3: Properties of recovered ECC algorithms.

<table>
<thead>
<tr>
<th>ID</th>
<th>(d_{\min}(cw))</th>
<th>(d_{\min}(data))</th>
<th>symbol size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD-1</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Intel-1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.4: ECC error handling software with a default Debian 9.

<table>
<thead>
<tr>
<th>ID</th>
<th>OS log</th>
<th>Firmware log</th>
<th>Crash on UE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD-1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Intel-1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 5.5: Error patterns that can circumvent ECC.

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern</th>
<th>Config.</th>
<th># flips</th>
<th>Flips location</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD-1</td>
<td>([P_1])</td>
<td>Ideal</td>
<td>3-BF-16</td>
<td>3 symbols, 1 in control bits</td>
</tr>
<tr>
<td>AMD-1</td>
<td>([P_2])</td>
<td>Ideal</td>
<td>4-BF-16</td>
<td>Min. 2 symbols</td>
</tr>
<tr>
<td>Intel-1</td>
<td>([P_3])</td>
<td>Ideal</td>
<td>4-BF-8</td>
<td>Min. 2 symbols</td>
</tr>
<tr>
<td>Intel-1</td>
<td>([P_4])</td>
<td>Default</td>
<td>2-BF-8</td>
<td>Min. 2 symbols</td>
</tr>
</tbody>
</table>

Table 5.6: Percentages of rows with corruptions in an ECC DIMM.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>([P_1])</td>
<td>0.12%</td>
</tr>
<tr>
<td>([P_2])</td>
<td>0.12%</td>
</tr>
<tr>
<td>([P_3])</td>
<td>0.06%</td>
</tr>
<tr>
<td>([P_4])</td>
<td>0.60%</td>
</tr>
</tbody>
</table>
Table 5.7: Percentages of rows with corruptions in the flip database of Tatar et al. with 14 DIMMs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Bit flips</th>
<th>$[P_1]$</th>
<th>$[P_2]$</th>
<th>$[P_3]$</th>
<th>$[P_4]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>200468</td>
<td>18.38%</td>
<td>04.41%</td>
<td>00.79%</td>
<td>29.51%</td>
</tr>
<tr>
<td>$A_2$</td>
<td>21542</td>
<td>00.23%</td>
<td>00.03%</td>
<td>00.03%</td>
<td>02.81%</td>
</tr>
<tr>
<td>$A_3$</td>
<td>2926</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.30%</td>
</tr>
<tr>
<td>$A_4$</td>
<td>256359</td>
<td>26.80%</td>
<td>08.52%</td>
<td>02.10%</td>
<td>37.52%</td>
</tr>
<tr>
<td>$B_1$</td>
<td>1504</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.00%</td>
</tr>
<tr>
<td>$C_1$</td>
<td>16489</td>
<td>00.09%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>01.32%</td>
</tr>
<tr>
<td>$D_1$</td>
<td>2131</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.66%</td>
</tr>
<tr>
<td>$E_1$</td>
<td>202630</td>
<td>06.30%</td>
<td>00.76%</td>
<td>00.14%</td>
<td>17.16%</td>
</tr>
<tr>
<td>$E_2$</td>
<td>24587</td>
<td>00.06%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>01.51%</td>
</tr>
<tr>
<td>$F_1$</td>
<td>413796</td>
<td>51.09%</td>
<td>26.02%</td>
<td>06.00%</td>
<td>53.03%</td>
</tr>
<tr>
<td>$G_1$</td>
<td>15990</td>
<td>00.06%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.93%</td>
</tr>
<tr>
<td>$H_1$</td>
<td>16087</td>
<td>00.03%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.77%</td>
</tr>
<tr>
<td>$I_1$</td>
<td>130187</td>
<td>00.82%</td>
<td>00.03%</td>
<td>00.00%</td>
<td>06.24%</td>
</tr>
<tr>
<td>$J_1$</td>
<td>7185</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.00%</td>
<td>00.70%</td>
</tr>
<tr>
<td>AVG</td>
<td>93705</td>
<td>7.42%</td>
<td>2.84%</td>
<td>0.65%</td>
<td>10.89%</td>
</tr>
</tbody>
</table>
5.6 Challenge $C_2$: ECC-aware Rowhammer

This section addresses $[C_2]$ and shows how an attacker armed with details on the ECC function can reliably trigger Rowhammer bit flips that bypass ECC memory with no crashes. To this end, we show an attacker can observe bit flips using a side channel and then control bit flips using carefully selected data patterns in memory.

5.6.1 Observing bit flips

We now present a novel side channel that allows an attacker to observe bit flips that trigger correctable ECC errors. For this purpose, we use double-sided Rowhammer (i.e., accessing two aggressor rows targeting a victim row in between) to trigger bit flips and then measure the number of clock cycles it takes to access the victim row. On setup Intel-1, we select 3K aggressor-victim pairs and measure the DRAM access time on the victim row after Rowhammer. In case of a bit flip in the victim, this access triggers a correctable ECC error. We also randomly select 3K pairs that are potential targets for Rowhammer (i.e., map to adjacent rows), but that do not trigger any error after Rowhammer. To confirm ECC error correction is triggered, we read platform-specific hardware registers that record the presence of an ECC correctable error.

Figure 5.6 shows that accesses to data triggering correctable ECC errors are slower than those to data with no bit flips. The timing difference is three orders of magnitude, yielding a reliable timing side channel to distinguish between the two cases. Furthermore, we note that, in the error case, the access time has higher dispersion compared to the error-free case.

To show this side channel is present on different platforms, we target a single vulnerable aggressors-victim pair across our setups. In this experiment, each pair is hammered in two rounds each comprising 100 Rowhammer iterations. In the first round, we choose data such that errors are triggered. In the second round we change the data such that no errors are triggered. On setup Intel-1, we confirm the error case is slower by a factor of 563.1x compared to the error-free case. On setup AMD-
CHAPTER 5. ECC MEMORY AND ROWHAMMER ATTACKS

Figure 5.7: ECC memory access times for all the 8-byte chunks in 5 victim rows. The peaks correspond to bit flip-induced ECC errors corrected by hardware.

1, however, we observe a difference of only a factor of 1.01x. To closely examine the latter scenario, we randomly pick 5 vulnerable victim rows, hammer them, and measure the DRAM access time for each 8-byte word in the victim row. We repeat this experiment 100 times per victim row and report the average access time in Figure 5.7. As evident by the peaks in the figure (marking synchronously corrected ECC errors), even a minimal difference in the number of cycles to access the victim row is sufficient to reliably distinguish error from error-free cases. Interestingly, we also observe that, in some cases, error accesses are faster than error-free ones. Such negative peaks (first and fourth subplot in Figure 5.7) seem to only occur in the case of 0-to-1 bit flips. We leave the study of this phenomenon as future work.

In summary, the presented side channel is reliable enough to observe bit flips triggering ECC error corrections. Moreover, the side channel can reveal the exact location and direction of the bit flip. In the following, we investigate the source of the side channel in hardware and software.

ECC error handling architecture. ECC error detection is synchronous with respect to a given memory access. In particular, in response to a memory access request from the CPU, the memory controller immediately retrieves the data and its associated ECC bits from memory. Before returning the data to the CPU, the controller checks the data for errors. Note that, when so-called scrubbing is enabled, the controller can also periodically check the memory for errors with no CPU synchronization. However, given the low scanning frequency (a few hours for a full memory scan), its impact can be safely ignored for our purposes (short-lived Rowhammer attacks).

Once an error is detected by the memory controller, error-correcting operations are immediately performed by the hardware. Since the hardware has to correct (and to write back) the data via a slow path, this may introduce a measurable latency on the corresponding memory access and give rise to a timing side channel. In addition, the hardware needs to inform the system of the event using one of the following options (depending on the boot-time configuration): raise an exception at the software level or invoke a system management interrupt (SMI) handler.
With the first option, a machine check exception (MCE) is triggered as soon as the error is detected—even if interrupts are disabled \cite{194}. With a failing memory cell, correctable machine check interrupts (CMCIs) become frequent, resulting in non-trivial system overhead due to excessive time spent servicing interrupts. To reduce the overhead, an OS driver may dynamically switch to polling mode, where CMCIs are blocked and error accounting registers are polled explicitly. In both cases, errors are logged inside the OS and, depending on the OS configuration, the memory page containing the error is masked, the system is restarted, or the faulting process is killed \cite{195}. However, the OS does not have accurate knowledge of the physical location of the error (e.g., the exact DIMM, DRAM address, etc.), which makes it hard to implement sophisticated error handling policies.

This problem is solved with the second option, where an SMI handler can use platform-specific information to recover the exact physical location of the error. This information can then be saved in Advanced Configuration and Power Interface (ACPI) tables or other error-reporting registers. To inform the OS of the event, the SMI handler ultimately raises an MCE. This option is widely used on recent Intel Xeon machines and it is known as Enhanced Machine Check Architecture \cite{196}.

In both cases, a software chain that involves expensive operations is synchronously executed as soon as an error is triggered in response to a given memory access. This may introduce significant access latency and give rise to another timing side channel to detect ECC correctable errors.

**ECC error handling in practice.** As evidenced earlier, ECC error handling side channels may originate from both hardware and software operations. We now revisit our earlier experiments across setups to exemplify their availability on real-world platforms in their default configurations.

On setup AMD-1, uncorrectable errors crash the system. Correctable errors are reported by the OS driver and appended to a dedicated MCE log file (other than being logged at the firmware level). These synchronous software operations are lengthy and give rise to the strong timing signal we observed in Figure 5.6. Had an SMI handler been enabled in our setup, the signal would have been even stronger, given that studies show that handling an SMI is up to 171x times slower than simply triggering an MCE \cite{186}. In addition, we observe that, by default, on the Debian 9 distribution (Linux kernel 4.9.3) used in our setup, the MCE log file is world-readable, yielding an even more convenient side channel to observe bit flips.

On setup Intel-1, uncorrectable errors do not crash the system. In addition, the available OS driver recognizes the memory controller but does not report correctable errors. In other words, no MCE event is logged by the OS. Correctable and uncorrectable errors are logged in a firmware log, but only after a certain threshold is reached. While no logging or other software/firmware operations take place in the common case, the error handling operations performed by the hardware at memory access time are still sufficiently lengthy to give rise to the crisp timing signal we observed in Figure 5.7.

In summary, while ECC-equipped platforms may be configured in several different ways, error correcting operations carried out in hardware or software are consistently

\footnote{\texttt{/var/log/mcelog}}
observable across platforms through a variety of side channels. This allows attackers to reliably observe bit flips as a prelude to end-to-end Rowhammer attacks on ECC-equipped platforms.

5.6.2 Controlling and composing bit flips

It has been long known that Rowhammer bit flips are data-dependent. For example, the original Rowhammer paper [160] showed that a stripe pattern in DRAM’s array-of-rows organization (even/odd rows populated with 0s/1s or vice versa) induced the most errors. Since then, similar patterns have been used to maximize the number of bit flips and ease Rowhammer exploitation. We now aim to show that such data-dependent behavior can also be used to control and compose bit flips and enable ECC-aware Rowhammer exploitation. We start with showing how data patterns can be used to enable/disable individual bit flips and later show such behavior is independent of neighboring flips or data patterns enabling composability.

Controlling individual bit flips. We start by exhaustively testing our memory chips using double-sided Rowhammer with 4 possible data patterns: (i) 0/1-stripe (aggressor rows populated with all 0s, victim rows populated with all 1s), (ii) 1/0-stripe (aggressor rows populated with all 1s, victim rows populated with all 0s), (iii) 0-uniform (aggressor and victim rows populated with all 0s), and (iv) 1-uniform (aggressor and victim rows populated with all 1s). Across our setups, we observe numerous bit flips in the two stripe configurations and no bit flips in the uniform ones. To confirm the latter result, we progressively reduce the DRAM refresh rate until we observe bit flips for the uniform patterns. This only happens for unstable system configurations with very low refresh rates, where bit flips occur even without Rowhammer.

This experiment empirically shows an important property of Rowhammer: bit flips occur due to parasitic current [197], which induces capacitors storing opposite electric charges (i.e., data values) to interfere with one another and cause charge leakage in the victim cells. The direction of the bit flip ($1 \rightarrow 0$ vs. $0 \rightarrow 1$) triggered by a particular stripe pattern (0/1-stripe vs. 0/1-stripe) is an artifact of data scrambling operated by the memory controller, which stores 0s (or 1s) as a charged (or non-charged) state. However, since scrambling on commodity systems operates by XORing data values with an address-dependent bitmask that repeats consistently across (adjacent) rows [188], the bitwise stripe pattern is preserved even in the presence of scrambling. In other words, for every bit $i$ in a given aggressor-victim-aggressor row tuple, data scrambling can (if at all) turn a $0 \rightarrow 1$ bit column (assuming 0/1-stripe) into a $1 \rightarrow 0$ column (and vice versa), but always preserve the stripe (or in other cases uniform) pattern at the bit granularity. This property shows that, somewhat counterintuitively, we can ignore data scrambling to control Rowhammer bit flips with (stripe) data patterns. It also suggests we can enforce bit-granular stripe patterns to control individual bit flips.

To confirm this intuition, for each bit flip triggered in the previous experiment, we flip the corresponding (column-wide) bits in the aggressor rows to enforce a bit-granular uniform pattern and hammer again. Across our setups, we observe this is consistently sufficient to disable the original individual bit flips. Restoring the original bit-granular stripe pattern consistently re-enables every given bit flip. This experiment
shows we can reliably control individual bit flips. In other words, for every bit $i$ in a given aggressor-victim-aggressor row tuple, setting aggressor bit values to enforce a column-wide uniform pattern ($0 - 0 - 0$ or $1 - 1 - 1$) prevents occurrence of any flips in the victim bit, while setting aggressor bit values to enforce a column-wide stripe pattern ($0 - 1 - 0$ or $1 - 0 - 1$) induces flips in the victim bit (assuming the underlying cell is vulnerable). We can then switch between the two patterns to selectively enable/disable individual bit flips.

**Impact of neighboring bit flips.** We now have the ability to control individual bit flips starting from a given data pattern configuration in an aggressor-victim-aggressor row tuple. We now want to verify whether controlling multiple bit flips in the same ECC word at the same time is viable. This property is necessary to ensure composability of bit flips and is only realistic with no cross-bit-flip interference. To confirm the absence of such interference, we select all the victim ECC words that revealed multiple stripe-induced bit flips in our previous experiment, and exhaustively test all the relevant combinations of aggressor bit values. For example, given a victim ECC word with only two bit flips at offset $i$ and $j$ with the $0/1$-stripe pattern, we test the 4 possible combinations of column-wide $0/1$-stripe (or $1/0$-stripe) at offset $i$ and column-wide $0/1$-stripe (or $1/0$-stripe) at offset $j$. We say that there is no cross-bit-flip interference in a given victim ECC word iff the bit flip $i$ ($j$) is solely dependent on the aggressor bit values at offset $i$ ($j$). Across our setups, we observe no interference in any vulnerable ECC word, empirically confirming we can control multiple bit flips at the same time in a given word.

**Impact of neighboring data.** Our last experiment showed we can control individual bit flips with no interference from neighboring bit flips nor neighboring aggressor bit values. This was the case even for adjacent bit flips, showing that value changes in the aggressor bits at offset $i + 1$ (or $i - 1$) have no impact on a bit flip at offset $i$. To achieve fully unconstrained bit flip composability, however, we also need to study the impact of neighboring data values in the victim row.

For this purpose, we set up a new experiment, in which we select all the aggressor-victim-aggressor row tuples that trigger a single bit flip and randomly assign them one of the following data patterns: $D$ (column-wide $1/0$-stripe pattern in the bit flip location, random values elsewhere in the aggressor rows, and 0s elsewhere in the victim row) and $N$ (same as $D$, but 1s are used elsewhere in the victim row). The patterns are designed to stress the extreme cases of data values following (or not following) the direction of the bit flip (respectively). For this reason, we present results with data scrambling disabled, but we observed a similar trend with data scrambling enabled.

Figure 5.8 presents our results, depicting the probability distribution of the difference between the number of bit flips induced by $D$ and $N$ patterns as a function of the probability of the occurrence of the $D$ pattern (which we vary in every experiment). As the difference is generally less than 2.5% across setups, this shows that even neighboring data values in the victim row have little or no influence on a given bit flip. This confirms an attacker can surgically manipulate aggressor data bits to obtain fully composable, data-controlled bit flips and target arbitrary victim data in a given ECC word.

Interestingly, in some setups (e.g., AMD-1), there seems to be less interference,
showing that, while the properties we described well-approximate DRAM behavior across setups, they cannot perfectly model all the physical constraints in general. However, our approximations are sufficient to reliably mount practical attacks, as shown by our end-to-end exploit.

5.7 Challenge $C_3$: a practical ECCploit

In this section, we present ECCploit and show how an attacker—armed with knowledge of the ECC function, a side channel to observe bit flips, and the ability to control/compose bit flips via data patterns in aggressor rows—can mount practical end-to-end Rowhammer exploits on ECC-equipped systems. ECCploit consists of three phases. First, we template memory to find correctable bit flips. Second, we try to combine multiple of these bit flips to create error patterns that the ECC function is unable to detect. Finally, we use these patterns to launch exploits on three different victims: page table entries [161], RSA public keys [165], and binary code [168].

5.7.1 Templating correctable errors

In the templating phase, we probe the memory to see if we can safely trigger bit flips using Rowhammer. In particular, we only want to cause errors that the ECC function can correct automatically. Although the error correction ensures that we cannot observe these bit flips directly, the side channel presented in Section 5.6 still lets us detect them.

Target address selection. Templating starts with a list of potential aggressor locations ($a_1$ and $a_2$ in the case of double-sided Rowhammer) and victim ($v$) addresses which should both map to the same bank but different (neighboring) rows. Obtaining this list is trivial if we know the mapping between virtual and physical addresses. In our exploits, we rely on existing reverse engineering techniques to reconstruct such
5.7. CHALLENGE $C_3$: A PRACTICAL ECCPLOIT

5.7.1 Pattern selection. Our attack uses double-sided Rowhammer to detect usable tuples of aggressor-victim-aggressor ($a_1, v, a_2$). To ensure a crash-free templating strategy (i.e., only triggering correctable ECC errors in vulnerable locations), we arrange values in aggressor and victim rows such that the Hamming distance is less than or equal to the number of errors $E$ that the ECC algorithm is capable of correcting. In other words, we make sure that for each ECC word in the victim row, the corresponding ECC words in the aggressor rows are only $E$ bit flips apart. Assuming $x$ is the value stored in an ECC word, and $x'$ is the value with $E$ bits flipped, we can either store $x$ in the victim ECC word and $x'$ in the aggressor ECC words or $x'$ in the victim ECC row and $x$ in the aggressor ECC words to check for correctable bit flips in either $1 \rightarrow 0$ or $0 \rightarrow 1$ directions due to the resulting striping patterns.

5.7.2 Search strategy. Rather than targeting a single ECC word and single word offset for each Rowhammer trial, we target all the words in the victim row at the same time during each hammering attempt. For each word, we consider a different set of $E$ bits in subsequent attempts. For instance, if the ECC corrects single bit errors, we hammer first with bit patterns in the aggressor and victim rows such that aggressors and victim differ only in the most significant bit of each of the ECC words in the row, then with patterns that differ only in the next bit, and so on. At each trial, we read from the entire victim row all at once and use our side channel to detect bit flips anywhere in the row—we found this is reliable even at the row granularity. This strategy exploits composability of bit flips and allows us to batch many independent tests and increase the templating efficiency. For instance, if the ECC corrects single bit errors, this strategy requires only as many trials per tuple as the number of bits in a single ECC word. ECC algorithms that use multiple-bit symbols (e.g., ChipKill) require even fewer trials as a row contains fewer symbols.

If we detect bit flip(s) anywhere in the victim row, we need to hammer the tuple a few more times to identify the flipping ECC word(s). For this purpose, we perform a (pseudo-)binary search—omitting stripe patterns in words we are not testing—until we reproduce the bit flip(s) on one or more words. The entire process is repeated twice for each tuple using the two possible stripe patterns. This is to identify vulnerable bits in both directions ($1 \rightarrow 0$ or $0 \rightarrow 1$). After scanning all the tuples in memory, we note down all the vulnerable 1-bit templates with the corresponding ($a_1, v, a_2$) tuple, the ECC word, the word offset, and the direction of the bit flip in the victim row.

5.7.2 Combining bit flips

Given our knowledge of the ECC algorithm and the 1-bit templates inducing correctable bit flips from the previous step, the goal of this phase is to combine multiple bit flips in a single ECC word and produce new words that escape ECC detection. As a first step, we group together all the 1-bit templates that have the same aggressor rows, victim row, direction, and ECC word in a template group.

Next, we generate possible flipped words that, when induced via Rowhammer, bypass the target ECC algorithm. Specifically, for every template group, we want to find a combination of $k$ 1-bit templates that would induce $k$ bit flips that result in a
corruption that ECC does not correct ($P_4$) or even detect ($P_1$, $P_2$ and $P_3$). For simplicity, the current version of ECCploit only targets flips in the data bits and not in the control bits. While this is enough for our setup, one can optimize ECCploit further to take control bits into account.

Figure 5.9 shows the results of our templating step on the Intel-1 machine. On this machine, we can directly observe (detectable) corruptions without crashing the system. When we cannot directly observe uncorrectable errors (e.g., AMD-1), we can instead use the side channels discussed in Section 5.6.1. Overall, we only have 265 templates available. When directly observing bit flips, it takes 4 hours to find these templates. Using the word-level side channel, it takes 6 days, and using the row-level side channel, it takes us 8 weeks to find these templates. To compare, assuming no ECC support, it would take us at most 1 minute to find 265 templates. This shows that ECC does significantly reduce the attack surface of Rowhammer attacks, by forcing the attacker to go through a much lengthier templating step. However, this is typically unimportant in practical attack settings, where the attacker can run code on demand on the victim machine and complete a templating step of hours or even days in complete isolation without interfering with the rest of the system. After templating is over, ECC has essentially no impact on the exploitation step, which completes in seconds or minutes similar to existing non-ECC exploits. Next we discuss how we use our templates to build practical exploits on ECC memory.
5.7.3 Exploitation

Armed with vulnerable ECC-aware templates, an attacker can now mount practical exploits by (i) massaging the target data onto the vulnerable location, (ii) setting the corresponding aggressor bit values as dictated by the templates, and (iii) and hammering to reliably reproduce the (composed) bit flips on the victim data. This exploitation strategy is similar, in spirit, to the one employed by existing reliable Rowhammer attacks [165]. The key difference—and challenge for ECC-aware exploitation—is that the number of useful templates is now much lower, given that we need a carefully-selected combination of bit flips to bypass ECC. Furthermore, unlike existing Rowhammer exploits, ECC templates corrupt multiple bits and this can complicate existing Rowhammer attacks.

To study the effectiveness of our ECCployt attack in real-world exploitation settings, we reproduce three existing Rowhammer attacks on Intel-1. (i) The original Rowhammer attack by Seaborn [161], which flips bits in page table entries (PTE) to map an unauthorized page (ideally a page table page) for privilege escalation, (ii) the attack introduced by Razavi et al. [165] which flips bits in a RSA key to compromise its cryptographic strength for authentication bypass, and (iii) the attack introduced by Gruss et al. [168] that flips bits in opcodes, leading to user authentication bypass in the sudo command.

Page Table Entry (PTE) ECCployt. Like the original attack by Seaborn et al. [161], we spray physical memory with page tables and then try to gain access to an inaccessible page by flipping a bit in a PTE. To implement this attack, we need to consider the format of the PTE. The format of the PTE can vary across different architectures. In modern Intel and AMD machines, PTEs are 64 bits wide and store the physical address of a page in bits 12 to $L$, where $L$ is the number of bits required to address the machine’s physical memory. Importantly, Intel requires that bits $L$ to 51 are zero, lest any access triggers a general protection fault which would crash the machine. AMD even prescribes a zero value for all bits between $L$ and 63. Given this, useful templates contain at least one bit flip between bit 12 and $L$ in 64 bits chunks and do not trigger a $0 \rightarrow 1$ bit flip in the $L:51$ range on Intel machines and $L:63$ range on AMD machines. Note that bit flips on the first 12 bits are often harmless (e.g., cacheable flag).

Results. From our discovered 265 templates, 6.15% are exploitable. The rest are templates that would crash the system because bits would flip in the reserved field of the PTE. As shown in Figure 5.9, we find the first suitable template after 19 minutes if we can directly observe the bit flips, and 12 hours or 4 days using the side channels respectively. Without ECC, it would take less than 2 seconds to find a suitable template.

Summarizing, even with an imperfect page table spraying strategy of the Seaborn attack, we were able to map unauthorized memory pages with a success rate of 39.9% and a page table page with a 2.5% success rate. In the remaining cases, the attack fails to modify any PTE of the attack process, but no crashes occur. By tracking the correctable error counters, we confirmed that when there is no change in the PTE, as either no bit flip occurs or ECC corrects the error. This happens because the victim PTE does not always have the target bits set in the direction of the chosen template.

Brasser et al. [192] report a 5% success rate in a similar non-ECC setting for...
mapping page table pages, which shows that our ECC-based exploitation strategy has relatively little impact on the success of the attack compared to traditional Rowhammer exploits. On our testbed, a more sophisticated massaging strategy such as the one employed by Drammer [166] can obtain a significantly higher success rate in mapping a page table page in the address space (39.9% in the ideal case).

**RSA ECCploit.** RSA [199] is a public-key crypto system which relies on the infeasibility of factorizing the product \(n\) of two large prime numbers with a similar number of bits. The attack uses the fact that a single-bit-faulted \(n\) \((n_1)\) is easy to factorize as the chance of the factors of \(n_1\) being of similar size is very low—the probability to efficiently factorize \(n_1\) is 12-22% [165]. We claim that in the presence of \(t\) bit faults \((t \geq 2)\), \(n_t\) is efficiently factorizable with at least the same probability as \(n_1\). This is because flipping a single bit versus flipping \(t\) bits in \(n\) only changes the quantity that is added or subtracted to \(n\). The result in both cases is a natural number with the same probability of being easily factorizable. Formally, using the Erdős-Kac [200] theorem, the number of distinct prime factors of \(n_1\) and of \(n_t\) follows the standard normal distribution with the mean and variance \(\log \log n\). Because \(n_t\) and \(n_1\) are of similar sizes, the probability to efficiently factorize the faulty \(n\) is the same in both cases—12-22%.

**Results.** To experimentally confirm this claim, we use 1337 randomly generated RSA keys from each size class of 1024 bit, 2048 bit and 4096 bit. We then replicate Flip Feng Shui [165] using our ECC templates. On average, our 265 templates could only mutate a given 1024 bit key 2.8 times, a given 2048 bit key 5.5 times, and a given 4096 bit key 9.4 times. Given a 1 hour cutoff time to ECM [201], we can factorize 45.1% of the 1024 bit keys, 37% of the 2048 bit keys and 28.7% of the 4096 bit keys. Without considering the factorization and memory deduplication delay, if we can directly observe the errors it takes us on average 2 hours, and 3 days or 4 weeks if we use the side channels as shown in Figure 5.9. Without ECC, it takes us less than a minute to achieve similar success rates.

**Opcode modification ECCploit.** This attack corrupts instructions in memory to bypass certain security checks [168]. As already mentioned, on ECC protected memory, more than one bit flip within the same ECC word is necessary to bypass the ECC protection. On synthetic x86_64 binaries that mimic authentications, we find that the probability of the code being successfully attacked slowly grows from 5% to 10% when the number of bit flips in 8 bytes increases from 1 to 4 respectively. On the other hand, the probability of the program to crash is 55% when 4 bits are changed as opposed to 20% when a single bit is flipped. To investigate whether corrupting opcodes is feasible with ECC templates in a real application, we target sudoers.so which is responsible for privilege elevation functionality provided by the sudo command.

**Results.** In the same version of the binary, Gruss et al. [168] find 29 candidate instructions in which a single bit flip yields unauthorized access. Template #36 flips bit 0 and 5 of a single byte, changing a conditional branch instruction \((jne \ $.8fa0 at offset 0xbdc0)\) to a \(mov\) instruction \((mov 0x1da(%rbp),%eax)\), leading to an authentication bypass. When observing ECC errors directly, we find this template in 32 minutes, and it takes 12 hours or 4 days when using the side channels as shown in Figure 5.9. Without ECC, we can target any of the 29 candidate instructions without worrying about crashes. We can find such a flip in 6 minutes.
5.8 Related work

Rowhammer. After the initial disclosure of Rowhammer [160], security researchers showed advanced Rowhammer-based exploitation of browsers [162, 163, 161, 202], clouds [165, 184] and mobile phones [167, 166], and even managed to flip bits across the network [202]. Although it was always clear that it is possible that more bits flip than an ECC function can handle, properly implemented ECC memory is still perceived as a practical mitigation for Rowhammer exploits [161, 168]. However, some researchers already questioned whether ECC is enough, and consistent with our findings, discovered that some systems do not always report ECC events [203]. We are the first to show that reliable Rowhammer attacks are possible, even if the system reports these events correctly.

Hardware reverse engineering. There are many undocumented features modern hardware systems. The complex hashing function that decides how physical addresses map to CPU cache sets is an example which is important for a variety of cache attacks [204, 205, 206, 207]. Maurice et al. [208] reverse engineer this mapping. DRAMA [198] reverse engineers the mapping function from physical addresses to DRAM addresses. Inside memory chips, each DRAM address is further decoded in banks, rows and columns. Jung et al. [209] reverse engineer this physical decoding scheme by applying a temperature gradient to memory chips. GPU architectures are sometimes undocumented, Frigo et al. [167] reverse engineer a common integrated GPU in mobile phones. In this chapter, we reversed engineered the ECC functions in common processors and used this to mount successful and reliable Rowhammer attacks.

ECC error handling and error injection. While others have studied the overhead of SMI handling [210, 186], the overhead of handling ECC exceptions is only briefly noted in the context of memory reliability [211, 164, 212]. Recently, Gottscho et al. [186] injected faults in memory with the help of a custom proprietary device and focused on the overhead of these errors. Instead, we proposed several new and cheaper ways to induce memory errors (e.g., a simple syringe needle probe).

5.9 Mitigations

We have shown that ECC alone is not an adequate Rowhammer mitigation. One way to strengthen ECC is to combine it with Target Row Refresh (TRR) [174]—another hardware mechanism, designed specifically to protect against Rowhammer. While there are reports of bit flips on memory with TRR [168, 166], we expect that a combination of ECC with TRR will make Rowhammer exploitation much harder.

State-of-the-art ECC algorithms in use today all target error patterns of off-the-shelf DRAM under normal conditions [164, 156] rather than adversarial cases. Another avenue for mitigations is to devise new Rowhammer-aware ECC algorithms that can be deployed either in hardware or software [213]. Moreover, to improve the guarantees of new ECC algorithms [214, 215, 216, 217], we may explicitly augment them with defenses against Rowhammer, either in software [218, 219, 220, 202, 192, 213] or in hardware—e.g., in the memory controllers or inside the memory chips them-
selves. As an example, in-DRAM ECC [221, 222, 223, 224], where the ECC engine resides inside each chip can co-exist with rank-level ECC implemented in the memory controller [224]. The in-DRAM ECC helps to mitigate Rowhammer, while potentially masking the side channel presented in this chapter (since the errors are corrected on die).

Another common solution against Rowhammer is to increase the DRAM refresh rate, but doing so wastes power. Also, the current trend in practice is exactly the opposite: manufacturers have started lowering the DRAM refresh rate to save power and relying on ECC for memory integrity [225, 226, 171, 222]. Since lowering the refresh rate dramatically increases the number of Rowhammer bit flips [160, 162, 163], doing so makes it easier to bypass ECC—we believe that it is time to reconsider such strategies in the Rowhammer era.

5.10 Conclusion

Rowhammer has evolved into a serious threat to computer systems from the smallest mobile devices to very large clouds, but so far machinery with high-end memory with error correcting code (ECC) has been free from such attacks. This has been due to the complex challenge of reverse engineering commodity ECC functions and, more importantly, to the narrow margins within which attackers must operate: multiple bits must flip in order to bypass the error correcting functionality, but flipping the wrong number of bits may crash the system. Thus, many believed that Rowhammer on ECC memory, even if plausible in theory, is simply impractical. This chapter shows this to be false: while harder, Rowhammer attacks are still a realistic threat even to modern ECC-equipped systems. This is particularly worrying, because all other existing defenses have already been proven insecure. Given the proliferation of Rowhammer vulnerabilities across a broad range of systems, we urgently need better defenses against these attacks.
We extended state of the art reverse engineering techniques but also applied reverse engineering to improve the security of embedded systems. The knowledge extracted through reverse engineering from firmware or from hardware, be it in the shape of vulnerability or just the understanding of how reliability mechanisms work, allowed us to showcase novel attacks and to motivate the need for developing new defenses of such systems.

In Chapter 2 we showed how to discover parsers and parser-like code in binary code. We also argued that this code is important for security researchers as it represents the first code that handles untrusted external input. To ease the analysis of binary code, in the process of building PIE we designed and open-sourced a binary to LLVM lifter. The binary lifter is able to recover high level constructions such as functions, function boundaries and even control flow graphs. PIE then uses static analysis on the recovered control flow graph to detect switch statements inside loops and other code constructions (features). Based on these features, we designed an automated ranking system to detect functions that contain parsers and parsing like function. We proved the importance of finding parsers and the viability of PIE by identifying a hidden boot loader menu on the PLC and an out-of-bounds memory access in a HTTP parsing code. With the help of dynamic analysis, we then showed how to trigger these vulnerabilities. We proved that even this lightweight analysis yields security oriented vulnerabilities (Q1.A).

In Chapter 3 we introduced JTR which aims to solve intricate indirect control flow (Q1.B) in firmware code. We first systematized how the compiler handles switch statements and then showed that in roughly 20% of the cases, jump tables are preferred over if-then-else constructions. These jump tables are one of the main sources of indirect control flow in benign firmware code. In addition, we presented how state of the art disassemblers that rely on pattern matching frequently mispredict the targets of these indirect control flows. This, in turn, leads to code being misclassified as data and as a result security solutions, which are based on binary rewriting, cannot secure these programs. JTR, being agnostic to the compilers and to the presence of debug information, is also able to predict the target of indirect transfers in hand written
assembly. In fact, we discovered one out-of-bounds control flow transfer in such code.

In Chapter 4, we showed that defending against hardware attacks at firmware level is a difficult task in practice. Specifically, we empirically determined that instruction duplication and other redundancy defenses are ineffective against fault injection through power glitching. Even more worrisome, we discovered that the information leakage through power side channel is amplified when instructions are duplicated. We estimated that it would take fewer trials for an attacker to extract secret information from the embedded device in this instance. Because we proved that the protection level offered by instruction duplication depends on the runtime configuration, reverse engineering plays an important role here (Q2). Specifically, static analysis can serve two purposes: to assess the runtime device configuration but also to check if instruction duplication is deployed at firmware level. We also released the compiler for instruction duplication to facilitate the research in this area.

In Chapter 5, we reverse engineered the inner working of error codes that are deeply embedded in the CPU, namely in the memory controller. Where the community once thought that ECC equipped systems were safe from Rowhammer attacks, we showed that these attacks are still possible and pointed out the guarantees offered by ECC. We proposed novel reverse engineering techniques that are based on bit flip injection and on cold boot attacks. We demonstrated that the bit flip needed for reverse engineering can be cheaply injected by using a syringe needle probe. Furthermore, the error reporting mechanism associated with ECC, designed to offer high reliability, introduces a memory access slowdown, that we leveraged in a novel proof-of-concept Rowhammer exploit (ECCploit). ECCploit works even in the presence of ECC memory, so we proved that relying solely on ECC as a protection against such attacks is not future-proof. Discovering and fixing this design flaw would not have been possible without reverse engineering the hardware itself (Q2). Our results accelerate the adoption of the next DDR technology which includes hardware defenses such as row access counting, but also motivate further research of other defenses.

Future directions

Maintaining the security of a system and embedded systems in particular is an ongoing race between defenders and attackers. While in the last decade defenders made significant progress in shielding and protecting these systems, the ever increasing number of new embedded devices poses a challenge for conventional defenses. Thus the arms race moves towards automation, that is automatic deployment of defenses and automatic exploit generation. In this dissertation, we analyzed and improved the automation of reverse engineering techniques to reveal security problems. This allows defenders to timely deploy countermeasures and fix these problems thus raising the bar for attackers. In this context, we have to continue the research at the intersection of reverse engineering and automation.

Automated analysis of embedded devices

In our work, we improved static analysis for binary code. However, in the process of reverse engineering we assumed access to firmware code. This assumption may prove
hard to satisfy in the future because recent technologies such as secure enclaves and trust zones encrypt the code. Therefore treating the embedded device as a black box is a valuable direction. Perhaps we can observe some effects of the computation that takes place in these black boxes. For example, in a fuzzing system, one idea is to rely on side channels (e.g., power measurements, timing differences) to track the code coverage and the progression of the fuzzer.

Even with access to firmware code, we still have challenges to solve. First, binary lifters and disassemblers are crucial components in the analysis of firmware code. However available tools in this area are not yet mature enough to handle the complexity of firmware code. In JTR we focused only on the indirect control transfers but the data-code interleaving problem remains partially unsolved. Second, while in the case of general purpose computer systems we can easily emulate and enable dynamic analysis (e.g., taint tracking, symbolic execution), in the case of embedded devices achieving emulation is challenging due to their highly specialized code (e.g., custom instructions, driver code for uncommon peripherals). To continue the fuzzer application, a hybrid analysis (static and dynamic), similar to PIE, may help identify error handling code and thus steer the fuzzer away from this code to achieve the same coverage faster.

**Automatic deployment of defenses**

The main disadvantage of programs coded in C language, that are widely used in the embedded world, is the lack of memory safety. As the price of compute power is decreasing, it becomes affordable to trade some resources such as memory footprint and speed for security. In fact, we observe initiatives to adopt other languages (e.g., Rust, Go, OCaml) which are statically typed checked and for which the compiler can detect out-of-bounds memory accesses at compile time. While not perfect, their adoption represents a forward leap in attack surface reduction because the developer must explicitly mark the “untrusted” code that directly interacts with the hardware in a potentially unsafe way.

Even if we showed in this thesis that automatic deployment of instruction duplication as a defense is not efficient against faults, we still believe that relying on compilers for hardening code against hardware attacks is a promising direction to follow. For this we have to redesign compilers such that they account for low-level hardware details, such as cache size, different fault injection models, or even the branch prediction behavior. Because some of these defenses incur costly overheads, as a first step we envision a semi-automatic system in which important resources are annotated by the developer and in which the compiler automatically provides the security guarantees for these resources.

Programs for personal computers already have a variety of mitigations that are automatically applied at compile time. These mitigations (e.g., canary values, stack cookies, various control flow integrity sanitizers) offer protection against trivial memory errors, but also against more advanced exploitation vectors (e.g., code reuse attacks, type confusion). Again, the decreased cost of computation power drives the adoption of the aforementioned mitigations in the embedded devices world. Unfortunately, we have yet to understand the behavior of these mechanisms in the presence of
recent hardware and micro-architectural attacks.

**The never-ending hardware reverse engineering**

In this thesis we presented merely a “snapshot” of hardware reverse engineering. But we believe that this is an ongoing process and that hardware reverse engineering will never cease to exist for two reasons.

First, reverse engineering itself enables other research directions that involve more reverse engineering. We can directly use the reverse engineering results. For example, to speed-up current memory safety solutions we can leverage the ECC properties reverse engineered in Chapter 5 and provide a red-zone implementation backed-up by hardware poison. But we also can *indirectly* use the reverse engineering work to reverse engineer other subsystems. For example, an indirect application of the reverse engineering of ECC is the study of memory accesses. More precisely, memory accesses caused by hardware features such as speculative execution or data prefetching that are at the center of a new class of attacks (i.e., Spectre, Meltdown and Foreshadow). Because these features are not publicly documented, studying them represents a reverse engineering effort. Currently, defenses are proposed against these attacks but we lack (to date) a complete picture of these features. As a result, such defenses are ineffective because they are treating an effect rather than the root cause. Building on our work, defenders now can use the hardware poison concept explained earlier, to reverse engineer the memory accesses caused by speculative execution. In this way, the extension of our reverse engineering work provides validation means of proposed defenses.

Second, the need for reverse engineering is still present even if hardware and systems are open and exhaustively or publicly documented. Hardware (e.g., CPUs, MCUs) uses some security through obscurity mechanisms. For example parameters of clock jittering, a defense against hardware side channel, or initial states for PRNGs are kept secret on purpose. As the security of critical devices is becoming more standardised we slowly move away from this model. Even if total openness were achieved, a state in which any security researcher would have access to the design of an embedded device, reverse engineering remains essential. First, through reverse engineering one can discover phenomena and behaviors that were not documented simply because it is impossible to predict the interaction between all the components of a complex system. Second, reverse engineering, both the methodology of applying it and the process itself, represents a valuable learning experience not only for the reverse engineer but also for the security community.
Contribution to papers and open source

The papers included in this thesis represent a collaborative effort. My personal contribution is as follows:

Chapter 2, PIE: Parser Identification in Embedded Systems. PIE uses a binary-to-LLVM translator, named bin2llvm<sup>1</sup>, that was originally designed and developed by me, Jonas Zaddach and Istvan Haller and extended by me. PIE is designed and implemented by me with valuable help from Jonas Zaddach. In addition, he performed the extraction of the GPS receiver firmware, the evaluation of PIE of the hard disk and contributed to the corresponding description in the paper. I evaluated PIE on the power meter, the PLC and on the GPS receiver and drafted the section from the paper that describes and evaluates PIE.

Chapter 3, JTR: A Binary Solution for Switch Case Recovery. Personal contribution to this paper includes: the full system design based on an improved version of bin2llvm, the evaluation and the draft version of the paper. The project is open sourced<sup>2</sup>.

Chapter 4, Instruction Duplication: Leaky and Not Too Fault-Tolerant! My contribution consists of the implementation of the instruction duplication compiler<sup>3</sup> the practical evaluation of the fault injection and of the side channel, with guidance and feedback from other authors. The side channel theoretical proof and analysis was performed and described by Kostas Papagiannopoulos. I drafted the first version and also finalized the paper.

Chapter 5, Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. I designed and performed all the reverse engineering

---

1<sup>bin2llvm</sup>: https://github.com/cojocar/bin2llvm
2<sup>JTR</sup>: https://github.com/cojocar/jtr
3<sup>LLVM-iskip</sup>: https://github.com/cojocar/llvm-iskip
work described in this paper and drafted the first version of the paper describing these attacks while receiving feedback from other authors.
References


[68] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C Necula. “XFI: Software guards for system address spaces”. In: OSDI’06 (cit. on p. 28).
[69] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Ben- 
net Yee, Karl Schimpf, and Brad Chen. “Adapting Software Fault Isolation to 
google.com/p/nativeclient/ (cit. on p. 28).

[70] Bin Zeng, Gang Tan, and Greg Morrisett. “Combining control-flow integrity 
and static analysis for efficient and validated data sandboxing”. In: CCS’18. 

[71] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bar- 
tel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 
“Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware 
taint analysis for android apps”. In: ACM SIGPLAN. 2014 (cit. on p. 28).

[72] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-Adaptive Mutational 
Fuzzing”. In: S&P’15 (cit. on p. 28).

detection via static taint and initialization analyses”. In: ACSAC’08 (cit. on 
p. 28).

Fuzzing”. In: ICSE ’09. Washington, DC, USA (cit. on p. 28).

Pipelined Symbolic Taint Analysis”. In: USENIX SEC’15 (cit. on p. 28).

[76] Sorav Bansal and Alex Aiken. “Binary Translation Using Peephole Superopti- 
mizers”. In: OSDI’08. url: http://dl.acm.org/citation.cfm?id=1855741.1855754 
(cit. on p. 28).

to compiler high IR in a binary rewriter. Tech. rep. University of 
Maryland, 2010. url: http://www.ece.umd.edu/~barua/high-IR-technical-
report10.pdf (cit. on p. 28).

[78] Matthew Smithson, K Anand, and A Kotha. Binary rewriting without reloca-
http://moflow.org/ref/Binary%20Rewriting%20without%20Relocation%20 
Information.pdf (cit. on p. 28).

[79] Mustafa M Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. 
“PMaC Binary Instrumentation Library for PowerPC/AIX”. In: Workshop on 
Bin. Inst. and App. 2006 (cit. on p. 28).

[80] Daniel Kästner and Stephan Wilhelm. “Generic Control Flow Reconstruction 
from Assembly Code”. In: SIGPLAN’02 (cit. on p. 29).

Binaries”. In: CAV ’08. Berlin, Heidelberg (cit. on pp. 29–34, 39, 42).

[82] Laune C Harris and Barton P Miller. “Practical analysis of stripped binary 
code”. In: ACM SIGARCH Computer Architecture News 33.5 (2005), pp. 63–68 
(cit. on p. 29).


REFERENCES


REFERENCES


[133] Pei Luo, Yunsi Fei, Liwei Zhang, and A Adam Ding. “Side-Channel Power Analysis of Different Protection Schemes against Fault Attacks on AES”. In: ReConf ig ’14 (cit. on p. 48).


REFERENCES


[157] S Satoh, Y Tosaka, and SA Wender. “Geometric Effect of Multiple-Bit Soft Errors Induced by Cosmic Ray Neutrons on DRAM’s”. In: (). IEEE. 2000 (cit. on pp. 65 [68].


REFERENCES


[186] Mark Gottscho, Mohammed Shoaiab, Sriram Govindan, Bikash Sharma, Di Wang, and Puneet Gupta. “Measuring the Impact of Memory Errors on Application Performance”. In: CAL’17 (cit. on pp. 74 [87] [95].

REFERENCES


REFERENCES


[208] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse engineering Intel last-level cache complex addressing using performance counters”. In: RAID’15 (cit. on p. 95).


Embedded devices are ubiquitous nowadays and interact with us on a daily basis. They are not only in charge of low impact actions like controlling household appliances or tracking your fitness level but also part of critical infrastructure such as traffic lights, water level control of dams or delivering electricity to your home. The large scale of their adoption, connectivity, and complexity are several reasons why securing these devices is challenging and remains an unsolved problem.

Reverse engineering is the process of understanding how a device or system works internally without accessing design documents or source code. Through reverse engineering, defenders and testers can discover vulnerabilities in these systems. This allows developers to fix the vulnerabilities before an attacker has the opportunity to exploit them, thus improving the overall security of these systems.

In the first part of the dissertation we explore reverse engineering aimed at firmware. Namely, first we improve reverse engineering by proposing a lightweight static analysis and show how to detect parsing and parser-like code inside binary code. Because this type of code deals with untrusted user input, a vulnerability located here has high changes of being exploitable and cripples the security of the device. Second, we propose a heavyweight analysis designed to solve intricate indirect control flow transfers inside firmware code. In firmware code, one source of these intricacies is C switch statements that are compiled to a jump table-based implementation in binary code. This leads to data and code interleaving which is a hurdle for reverse engineering tools such as disassemblers and decompilers, but also for binary security deployment tools.

In the second part of the thesis, we focus on a form of reverse engineering that is tightly coupled with the hardware. We show that software defenses such as instruction duplication are ineffective and even harmful in the presence of hardware attacks such as power glitching. Furthermore, the ineffectiveness of these defenses depends on the hardware runtime configuration which can be revealed by reverse engineering. Lastly, we reverse engineer the error correcting codes (ECC) embedded inside various popu-
lar memory controllers by extending fault injection attacks and leveraging cold boot
attacks. Initially an ECC equipped system was thought to be immune to Rowhammer
attack. With the knowledge gained through reverse engineering we show that this is
not the case, therefore exposing the dangerous false sense of security. In addition,
our result sprouts a new research direction focused on defenses against this class of
attacks.

In conclusion, in this dissertation we show not only how reverse engineering at the
boundary between hardware and software may help improve the security of computer
systems, but also advance the state of the art of hardware reverse engineering.