University of California
Santa Barbara

Analyzing and Securing Embedded Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Chad Samuel Spensky

September 2020

The Dissertation of Chad Samuel Spensky is approved.

Professor Timothy Sherwood

Professor Fabian Monrose, University of North Carolina at Chapel Hill

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

July 2020

Analyzing and Securing Embedded Systems

This work is dedicated to Dr. William Hunter Vaughan for inspiring me and cultivating my love for math and science.

Acknowledgements

Where to start... There are so many people that have helped, inspired, and pushed me to make this degree, and my career, possible. First, I need to acknowledge the obvious two — my biological parents — Steve Spensky and Linda Lovy, for bringing me into this world and supporting me throughout my various career decisions. I would also like to acknowledge my "pap pap," Ralph Mills, for showing me what hard work looks like (he worked three time jobs as a mechanic, a coal miner, and a crane operator while raising 12 children) and demonstrating the difference between simply living and living well — being surrounded by people who love and adore you. Next, is the man who taught me the meaning of family, inspired me to believe in myself, showed me what it means to be southern gentleman. Hunter Vaughan, my non-biological dad, has supported me in too many ways to count, and my life would be no where near as great as it has been without him in it.

My pursuit of this degree undoubtedly stemmed from a few of the life-changing teachers that I have interacted with throughout my education. I was a C student with little aspirations of post-secondary school, until a notable semester during my Junior year at Brooke High School. Peggy Walker (my Pre-Calculus and Trigonometry teacher) and Glenn Berkheimer (my Physics teacher) changed my view on education. They were the only teachers up to that point that let do Mathematics my way — mentally, without showing work, and very quickly — and supported my methods. For the first time in my life, I not only felt "smart," but knew that I was capable of great things. At the university level, Robert Heath (my Introduction to Theoretical Mathematics Professor) solidified my love for Mathematics. His brilliance, dry humor, and Mathematical creativity were awe inspiring — I still recite his "golf ball" problem to this day. Alexandros Labrinidis and Panos K. Chrysanthis introduced me to research, which played a significant role in my pursuit of a Ph.D., and facilitated my acceptance to the various graduate programs. Their guidance and mentorship was invaluable, and showed me, first hand, how much of a positive influence a great professor can be to a young, motivated student — I have been trying to pay it forward ever since.

I am forever grateful for Michael K. Reiter for taking a gamble on me and introducing me to Computer Security as a profession, versus the potentially-criminal activity that I was previously using it for. His mentorship, wisdom, and work ethic have made me the researcher that I am today — I still here Mike's voice every time I create a macro in a LaTeX document. I want to acknowledge Fred Brooks for showing me the true meaning of humility and being so generous with his time and knowledge, despite his legendary status in our field. I similarly hear Fred's wise words in my head when conducting research, giving presentations, and interacting colleagues — "Tell them what your going tell 'em, tell 'em, then tell them you told 'em." Fabian Monrose saved my career, and for that I feel forever indebted. I vividly recall our conversation over coffee where I told him, "I want to drop of graduate school." Fabian was able to help me work through my situation and lent a desperately needed ear and outside prospective. He would ultimately introduce me to Charles V. Wright, who then hired me at MIT Lincoln Laboratory as an intern and was later my supervisor and mentor when I joined the technical staff. Charles provided me with some of the greatest career advice of my life, and I still model my management style after him. I recite his words regularly, and even use them as a mantra in my life, "Any time that you can exchange money for time, do it." In context, we were discussing work, but these words echo deeper — we can always make more money; we cannot make more time.

I would also like to acknowledge some of my colleagues that inspired me to push myself professionally and helped me achieve the success that I have thus far. I want to acknowledge Samuel Kerr for keeping me honest on getting to work on time, "First!" and for being the first person to bridge my two worlds and become a great personal friend and an incredible colleague. Hongyi Hu is easily the most influential coworker that I have had thus far. Without him, my career at MIT LL would have been nowhere near as successful as it was — his impressive intelligence, methodic engineering principles, and calm demeanor were a necessary balance to my more gun-slinging style of conducting research. Similarly, Ben Nahill is one of the most interesting and inspiring people that I have met — he is uncomfortably smart, has a breadth of impressive talents, and is an all around great guy. Working with Ben always inspired me to push myself and to set evermore ambitious goals. Our Turning Machine is still the most fun I have ever had engineering anything. I need to acknowledge the amazing interns that have worked under me, Rick Housley and Kevin Leach. They kept me humble and pushed me to continue improving (because they were way smarter than me), showed me how rewarding it can be to share knowledge and success with others, and were large motivators for pursing this degree (*i.e.,* I realized how enjoyable mentoring and teaching was).

I am also indebted to the various managers that inspired me in my time at MIT LL. Eric Evans, the director of the lab, is another whose words frequently echo in my head, "20% of the things that you try should fail; otherwise you are not pushing the limits hard enough." I always remind myself of these words when failures inevitably happen — failures are necessary and expected, not something to be feared. My group leaders, Lee Rossey, John Wilkinson, and Roger Khazan have provided me with saint-like support and encouragement. Lee let me push the limits with my research and put full trust in me, despite my young age at the time, enabling me to produce some of the most fun and exciting work that I have ever done. John's support when I decided to return to pursue my Ph.D. was one of the greatest displays of altruism I have ever seen — it was clearly a loss to his team, but he was still eager to support my dreams. Roger's continued support and mentorship during my Ph.D. through the various projects that we have worked on

together has helped me become a “professional” and also helped keep my head up during the unavoidable tough times in graduate school.

Outside of my advisors, there are others who have unofficially co-advised me through this process. I have worked on so many projects with Hamed Okhravi and learned so much from him as a researcher at this point that it is hard to believe that there is no official “advisor” title connected to his name. Westley Weimer is another professor that has come to embody what an academic should be in my mind — absolutely brilliant, committed to advancing human knowledge, and always looking to pull those around him up. Hani Jamoon and Zhongshu Gu provided invaluable guidance during my time at IBM Research. My internship at IBM Research was during a low psychological point in my life and I am sure that my gratitude was not appropriately conveyed, but I will always cherish my memories of Yorktown Heights and the amazing people that I got to work with at IBM. Receiving the IBM Ph.D. Fellowship is still one of the highlights of my life, and came at time when I desperately needed some reassurance in myself as a person and as researcher; I am forever grateful and beyond proud to have received this award. Dick Kemmerer has been in the background helping me throughout me degree, providing guidance, historical insights, and plenty of good laughs.

I also want to acknowledge the Club Football team at the University of North Carolina at Chapel Hill. Easily, one of the greatest experience of my life was playing football under Coach Featherstone. Being a part of that team showed me what a brotherhood was, gave me infinite confidence in myself, and made a man out me. I would not be the person that I am today or have the work ethic that I do today without this experience. It is not every day that meet someone that truly, 100% loves what they do — Feathers did. He *loved* football and I am honored to have shared this passion with him and my teammates — “We all we got!” Similarly, I want to acknowledge all of the members of Shellphish. I thought I was good at hacking and understood computer security, until I met these guys. Again, the passion that they bring and their deep technical knowledge have always pushed me to learn more and be a better security practitioner.

I also need to acknowledge all of the friends that I have made throughout my life. There are far too many to list explicitly, but my childhood friends, everyone at Pitt, my fellow Semester at Sea "adventurers," my Tar Heel friends, Spike Yo' Drank and other Boston friends, and the incredible people that I have befriended in California, have made my life exceptional. I could not imagine any better friends, and it would have been impossible to achieve any of my success without their support.

Finally, I cannot express enough gratitude for Chris Kruegel and Giovanni Vigna, my advisors. They are the reason that I have a career at all in Computer Security at all. I met them in 2008 during a prospective graduate student visitation weekend, after being accepted to UCSB's Ph.D. program as a Bioinformatics student. I recall meeting Giovanni in the lobby and discussing his research on "underground economies," which I, unbeknownst to him, used to play a significant role in. Until then, I had never considered security as a career path. But I was instantly hooked and determined to study computer security for my Ph.D. Throughout my career, I kept tabs on the SecLab, spent some time with Giovanni in Saint Lucia at a conference, and would later have "the talk" about me leaving my job and pursuing a Ph.D. at UCSB during the Blackhat conference in Las Vegas. As they say, "the rest is history." My time in the SecLab, under their mentorship, has far surpassed my expectations. Giovanni and Chris are two of the most impressive people that I have met and the culture and community that they have created through the SecLab, iSecLab, and Shellphish is second to none. The people that I have met, the skills that I have acquired, and the day-to-day life in the SecLab were incredible and are memories that I will forever cherish. Indeed, I have already found myself copying many aspects of their management style and work culture in my own endeavors — they succeeded in creating something very special and I will do my best to share a piece of that with others.

I have always believed that life is journey and have always tried to enjoy the ride, wherever it may take me. I am fortunate enough to have lived a very full life thus far, experiencing both extremely low lows and extraordinary high, high points. Some of these high points were beyond my wildest dreams. The book of my life has been incredible thus far, and I cannot wait to see what future chapters hold for me. Receiving this degree is just another affirmation that anything is possible if you want it enough and are willing to work hard to make it happen. As Jay Z famously said,

"Difficult takes a day, impossible takes a week."

Curriculum Vitæ
Chad Samuel Spensky

Education

2020	Ph.D. in Computer Science, University of California, Santa Barbara.
2010	M.S. in Computer Science, University of North Carolina at Chapel Hill.
2008	B.S. in Computer Science and Mathematics, University Pittsburgh.

Publications

1. Chad Spensky, Aravind Machiry, Marcel Busch, Kevin Leach, Rick Housley, Christopher Kruegel, Giovanni Vigna. *TRUST.IO: Protecting Physical Interfaces on Cyber-physical Systems.* CNS 2020.
2. Aaron Mills, Donato Kava, Alice Lee, Chad Spensky, Stephen Eng, Michael Vai. *Trust, Assurance, and Protection for Microelectronics.* GOMACTech 2020.
3. Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Giovanni Vigna, Christopher Kruegel. *KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware.* Oakland 2020.
4. Bryan C. Ward, Richard Skowyra, Chad Spensky, Jason Martin, Hamed Okhravi. *The Leakage-Resilience Dilemma.* ESORICS 2019.
5. Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Davide Balzarotti, Yanick Fratantonio, Aurelien Francillon, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna. *Toward the Analysis of Embedded Firmware through Automated Re-hosting.* RAID 2019.
6. Kevin Leach, Ryan Dougherty, Chad Spensky, Stephanie Forrest, Westley Weimer. *Evolutionary Computation for Improving Malware Analysis.* GI Workshop 2019.
7. Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, Michael Franz. *PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary.* NDSS 2019.
8. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers Aravind Machiry, Chad Spensky, Jacob Corina, Nick Stephens, Christopher Kruegel, Giovanni Vigna
9. Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna. *BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.* NDSS 2017
10. Chad Spensky, Jeffrey Stewart, Arkady Yerukhimovich, Richard Shay, Ari Trachtenberg, Rick Housley, Robert K. Cunningham. *SoK: Privacy on Mobile Devices – It's Complicated.* PETS 2016.

11. Kevin Leach, Chad Spensky, Westley Weimer, Fengwei Zhang. *Towards Transparent Introspection.* SANER 2016.

12. Chad Spensky, Hongyi Hu, Kevin Leach. *LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis.* NDSS 2016

13. Andrew Weinert, Hongyi Hu, Chad Spensky, Benjamin Bullough. *Using Open-source Hardware to Support Disadvantaged Communications.* GHTC 2015.

14. Chad Spensky, Hongyi Hu. *Live Disk Forensics on Bare Metal.* OSDFCon 2014.

15. Lujo Bauer, Yuan Liang, Michael K. Reiter, Chad Spensky. *Discovering Access-control Misconfigurations: New Approaches and Evaluation Methodologies.* CODASPY 2012.

16. Michael K. Reiter, Vyas Sekar, Chad Spensky, Zhenghao Zhang. *Making peer-assisted content distribution robust to collusion using bandwidth puzzles.* ICISS 2009.

Patents

1. Hongyi Hu, Chad Spensky. Systems and Methods for Single Device Authentication. US Patent #US10182040B2.

Abstract

Analyzing and Securing Embedded Systems

by

Chad Samuel Spensky

Embedded systems (*i.e.,* single-purpose computers with tightly-coupled software and hardware) are now pervasive throughout in our increasingly digitized world. Due to the rapid growth of the embedded systems industry and the commercial pressure to implement new features, most of these systems are built using insecure hardware and have numerous latent software vulnerabilities. Unfortunately, the diversity of physical hardware and software implementations on these various systems along with their tight coupling between software and hardware have rendered most of our existing automated security analysis techniques ineffective. Attackers currently have the upper hand, as they need only discover a single vulnerability, whereas defenders must manually identify, and fix, *all* of the existing vulnerabilities. To make matters worse, many of these vulnerable embedded systems can interact with the physical world and, if compromised, could cause serious damage (*e.g.,* a public utility) or even death (*e.g.,* a medical device). To rectify this calamitous situation that we have created, we must be able to *1)* identify and fix problems with the existing systems that are already deployed and *2)* create future systems that are *fundamentally* secure, by design.

Embedded systems are more difficult to analyze than traditional computers because the hardware platforms that they run on are far more diverse, have strict hardware dependencies, are equipped peripherals that differ wildly between systems, and their execution typically depends on external phenomena that materialize as hardware interrupts. The depth of the analysis can be improved by developing novel hardware-based introspection techniques, which would provide analysts with the ability to observe the internal state of the *real* embedded system in real-world scenarios. The scale of the analysis can also be improved by decoupling the firmware from the hardware through emulation techniques, which would permit analysts to parallelize their analyses across numerous emulated systems, without the need for hardware, and also experiment with the embedded system in a zero-risk virtual environment. I developed a novel hardware-based introspection technique for embedded systems that provides real-time, high-level insights into modifications made to both volatile and non-volatile memory using a Field-Programmable Gate Array (FPGA) implementation and novel semantic-gap reconstruction techniques. I also developed an approach to support the decoupling of firmware from its hardware that can use either hardware- or software-based instrumentation of the system to record the hardware interactions on the real system and then convert these recordings into generalized, composable ω-automata that can be used in place of the hardware for emulation.

Embedded systems are also difficult to protect against hardware-based attacks, especially glitching. Ideally, firmware could be protected against these attacks using software-only techniques that could be deployed to the billions of existing systems to protect them from physical attacks, without physically replacing them. I developed an approach that permits embedded system developers to automatically inject various software-based glitching defenses into their code at compile-time, producing glitch-resistant firmware without the need for any code annotations or modifications to the embedded system's hardware.

Contents

Chapter 1

Introduction

I believe in a world where technology is used to enrich the human experience by alleviating monotonous tasks and providing access to resources that allow us to optimize our limited time on this earth. I have focused my research effort exclusively on topics that either make the human experience better (*e.g.,* easy-to-use security products) or help secure the foundation that we are building our society on (*i.e.,* fundamental changes to how we design and build computing systems). My approach to research is straightforward. First, I perform literature reviews and create tools to analyze existing systems to identify any bugs or fundamental design flaws. Then, with the results of the analyses, I either facilitate a fix for the current problem, if possible, or design a new system that is fundamentally immune to previously identified flaws.

Embedded systems (*i.e.,* single-purpose computers with tightly-coupled software and hardware) have surpassed commodity laptops and personal computers as the most pervasive computing devices. Thus, to ensure the security of our computing infrastructure, and the various activities that these systems perform, their security and reliability is critical. Currently, however; analyzing these systems is prohibitively difficult due to their diversity (both in hardware and software) and the hardware dependencies that render a majority of the existing analysis techniques ineffective. Therefore, a majority of the devices are on our worlds computing infrastructure are running unanalyzed software, which are likely riddled with yet-to-be discovered vulnerabilities. The sheer number of embedded devices connected to the Internet could spell disaster if they were used maliciously (*e.g.,* the Mirai botnet was used to cripple popular internet services for hours [1]).

This arms race, and the current state of security for embedded systems, has been seen before. In fact, it almost exactly parallels the situation from a decade or two earlier with malware on commodity computers. Researchers were ultimately able to subdue many of the widespread threats by increasing their automated analysis capabilities (*e.g.,* virtual machines [2–5] static analysis [6–8], and fuzz testing [9–11]) to understand what the attackers were doing, and deploying novel defenses to thwart them, *e.g.,* stack canaries [12], address space layout randomization (ASLR) [13], and control-flow integrity (CFI) techniques [14]. While our current situation with embedded systems is certainly more difficult, it is by no means intractable, and is even more serious, as the cyber-physical nature of many of these systems escalates the threats from a digital annoyance to real-world threats of physical damage or bodily harm. Thus, in my research, I have been examining techniques to both analyze existing, already deployed, embedded systems, as well as creating mechanisms to better defend the embedded systems of the future.

The difficulties associated with analyzing embedded systems stem from four factors: *1)* their software is heavily dependent on their hardware (*i.e.,* it expects specific hardware responses), *2)* the hardware and software are incredibly diverse, requiring significant upfront effort to adapt existing analysis techniques, *3)* much of the software is interrupt-driven, and *4)* many embedded systems are built using non-traditional or custom instruction set architectures (ISAs). All of the challenges compound to create a perfect storm of complexity, which hinders existing static and dynamic analysis techniques. In an attempt to address this problem, there have been two prominent forms of analysis for embedded systems: hardware-in-the-loop techniques, where the hardware is used either directly [15, 16] or as an oracle for hardware interactions [17–20]; and emulation, or re-hosting, techniques, which aim to completely remove the hardware and create a software-only environment for analysis [21–23]. In my research, we have contributed significantly to both areas. We created an introspection system, Low-observable Physical Host Instrumentation (LO-PHI) [24] that is capable of reading the memory and monitoring hard disk interactions of a running system, without any software or hardware modification, by using a custom-designed Field-Programmable Gate Array (FPGA) that can perform Direct Memory Access (DMA) and interpose the serial advanced technology attachment (SATA) interface (*Chapter* 3). To demonstrate the power of LO-PHI, which can be used to introspection *any* system that uses peripheral component interconnect (PCI), PCI express (PCIe) or SATA, we evaluated the most evasive software possible — environment-aware, evasive malware. Evasive malware is specifically designed to detect that it is being analyzed and defend itself accordingly. Thus, LO-PHI's ability to analyze such software definitively shows its ability to analyze *any* software on real hardware.

While powerful, LO-PHI is limited to introspecting higher-end systems that have a PCI or PCIe bus and use SATA-based storage, which excludes a majority of the Internet of things (IoT) devices that are currently flooding market. To address these more ubiquitous devices, we have developed a system, called CONWARE, for modeling and emulating embedded hardware peripherals. CONWARE does this be first interacting with the real peripheral on a similar platform and recording the low-level interactions. These recordings can then be modeled and used in emulation to enable scalable dynamic analysis, without the need for *any* hardware (*Chapter* 4).

My ultimate goal, however, is not to simply highlight the inadequacies of our current ecosystem, but to build a better future for our society. Indeed, I have focused a significant amount of my research effort on, what I believe to be, the biggest need in security, usable authentication. To this end, we have developed a ubiquitous authentication solution, Single Device Authentication (SDA), which has undergone numerous user studies, was successfully patented [25], and is the core technology of a company that I recently founded, Allthenticate. To facilitate this effort, and my dream of a better tomorrow, numerous technological advances were required. First, we wanted to make sure that the smartphones that we would be deploying our credentials on, and using as our interface with the users, are, in fact, secure. Thus, we performed an extensive study on the security mechanisms in place on these devices [26], specifically TrustZone and the embedded secure element (SE), which SDA uses to protect its credentials and interactions from malicious software. During this analysis, we identified a few fundamental flaws [27], various bugs [28, 29], and some technical shortcomings of the existing devices (*e.g.,* developers cannot easily leverage TrustZone on existing Apple and Android devices).

The goal of SDA is to authenticate to *everything* (including embedded devices, commodity computers, and web-based resources) in a distributed way. Thus, the security of the embedded devices becomes just as important as the smartphones that they are authenticating with. More precisely, many of the embedded devices in our authentication ecosystem are responsible for physical, security-critical operations (*e.g.,* unlocking a door or starting a car) and must be protected from remote, software-based attacks. To protect these critical interfaces, we developed TRUST.IO [30], a TrustZone-based protection that is capable of transparently protecting *all* physical interfaces with limited source code modifications (*i.e.,* the insertion of one function call and the implementation of a single callback function).

The security of TRUST.IO, SDA, and any secure computing system for that matter, necessarily depends on an uncompromised boot process to ensure that all of the software- and hardware-based security mechanisms are correctly initialized. In our current mobile and embedded world, where attackers frequently have physical access to the device, this requirement becomes even more stringent, requiring both physical (*e.g.,* anti-tamper) and digital (*e.g.,* signed code) deterrents. Even if all of these protections are implemented ideally, there is still a class of attacks, known as hardware glitching, that is capable of compromising a secure boot process [31–40]. These hardware-induced faults, or *glitches*, are capable of strategically disrupting the physical state of the processor such that instructions can be effectively "skipped" or data manipulated in nefarious ways. Glitching attacks have been used to compromise numerous commercial products (*e.g.,* the XBOX 360 [34], Playstation 3 [35], Playstation Vita [36], and Nintendo Switch [37, 38]), and enterprise Internet protocol (IP) phones [39]. Moreover, glitching has even been leveraged to bypass both Intel's Software Guard Extension (SGX) protections [41] and ARM's TrustZone [42] and even extract hardware-embedded cryptographic keys [40]. Thus, I have dedicated a significant amount of my research to both understanding and defending against these powerful attacks (*Chapter* 5). To facilitate our analysis, we created a framework that is capable of analyzing glitching attacks using both emulation and real hardware devices. We also analyzed the efficacy of various proposed and novel defenses through the development of GlitchResistor, the first automated, software-based glitching defense tool. We found that our software-only defenses were capable of completely eliminating single-glitch attacks (*i.e.,* attacks that only affect one clock cycle) and reducing the success rate of multi-glitch attacks by more that 56×, with a detection rate of over 99%. Indeed, this result is impressive since completely eliminating glitching attacks in practice is impossible without hardware modifications.

Analysis of complex embedded systems execution on their original hardware is now possible using LO-PHI, a novel hardware-based introspection technique, or in a fully emulated environment using Conware, a novel record-and-replay technique that is capable of generating high-fidelity automata from low-level hardware recordings. Similarly, defending these critical systems against physical glitching attacks is now tractable and quantifiable using GlitchResistor, a source code and compiler instrumentation tool that enables both researchers and practitioners to easily create glitching-resistant code for embedded systems. Yet, it is clear that *securing any embedded systems is not possible with software alone* — embedded systems fundamentally rely on the hardware that they run on, which must be leveraged for any effective defense or analysis. Hybrid security defenses must be adopted where the hardware and software are both leveraged to fully defend against attacks, either by integrating hardware-based security mechanisms into the software (*i.e.,* Trust.IO) or supplementing hardware-security features with novel software defenses (*i.e.,* GlitchResistor). Similarly, any analysis of embedded system software must necessarily use the hardware, either by instrumenting it directly (*i.e.,* LO-PHI) or by interacting with hardware to enable emulation and analysis in its absence (*i.e.,* Conware).

In my future work, I plan to explore every aspect of our embedded, ubiquitous computing society that we are creating, and to continue developing new technologies where they are needed (*e.g.,* re-architecting hardware to eliminate memory corruption attacks, designing usable security systems, and creating secure-by-design systems that require no cooperation from users or developers). While the world may never be free of cyber threats or undue technological burden, we can always strive to do better, and I plan to use my research and my technical skills to help move our society in a positive direction.

1.1 Permissions and Attributions

1. The content of chapter 3 is the result of a collaboration with Hongyi Hu, Kevin Leach, Brendon Chetwynd, Charles V. Wright, Joshua Hodosh, Ryan Whelan, Lee Rossey, Doug Stetson, and John Wilkinson, and has previously appeared in the

2016 edition of the Network and Distributed System Security Symposium.

2. The content of chapter 4 is the result of a collaboration with Aravind Machiry, Graham Foster, Colin Unger, Evan Blasband, Nilo Redini, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna.

3. The content of chapter 5 is the result of a collaboration with Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna, and is currently in submission to the the 2020 edition of the Annual Computer Security Applications Conference.

Chapter 2

The State of Embedded Systems Security

The field of computer science is currently undergoing a fundamental shift from personal computers (*e.g.,* laptop and desktop computers) being the pervasive computing device to smaller, embedded systems (*e.g.,* smart homes, computer-based automobiles, and automated industrial infrastructure). This paradigm shift has brought with it a similar shift in the computer security industry, both in what attackers are targeting and what those attacks are capable of. Long gone are the days when malware was written by ornery teenagers in their basements. The modern cyber security battlefield is filled with nation-state actors that are capable of not only accessing data but also affecting physical systems, potentially causing damage or bodily harm. For example, the proliferation of internet-connected embedded systems has enabled attacks against automobiles [43, 44], critical infrastructure [45,46], and medical devices [47–49]. The game has clearly changed.

Indeed, throughout my career, while the focus of my work has always revolved around embedded systems, the applications of my research has shifted. While LO-PHI is capable of analyzing any software, the research community at the time (only five years ago) was still focused on malware, and thus, LO-PHI's efficacy was demonstrated on malware, instead of a robot, for example. Nevertheless, here we are, less than a decade later, and publications that focus solely on malware are almost non-existent — and justifiably so. The threat of malware has been seriously diminished, facilitated by large-scale analysis, which was enabled through emulation and automation, and novel defense techniques. Unfortunately, embedded systems are now undergoing a similar cat and mouse game that malware once faced: the attackers have the upper hand and every new defense only seems to delay the next attack, but not fully prevent it. This is largely due to the inability to analyze these critical systems efficiently. Currently, embedded systems analysis typically entails significant manual effort by a human analysts and in most cases requires access to the physical hardware itself. Attackers need only find one single vulnerability that can exploits while defenders must find, and patch, all of the latent bugs — the cards are stacked in the attackers favor.

Embedded systems are more difficult to both analyze and defend than traditional computers because a few fundamental differences:

- They run on large variety of architectures (*e.g.,* ARM, MIPS, PIC, or AVR) and, in many cases, even have custom ISAs, which is contrary to the traditional computers that are almost exclusively one of two architectures: x86 or ARM.

- Most embedded systems have strict hardware dependencies, which rarely used well-defined software interfaces and are also frequently customized for the specific system or application, whereas traditional systems use standard hardware abstraction layers (HALs).

- The number of peripherals that embedded software interfaces with, and depend on, is almost uncountable (*e.g.,* sensors, motors, or communication interfaces) making manually implementing all of them intractable, unlike traditional computers, which have only a few predictable peripherals (*e.g.,* keyboard, mouse, screen, and hard drive).

- Embedded code is typically written to be interrupt driven (*i.e.,* external phenomena decides which code is executing), which means that simply executing the code is unlikely to yield any valuable insights, unlike malware, which is typically a self-contained payload that will execute to completion without external input.
- Finally, attackers typically have physical access to embedded systems, which changes the threat model significantly. Embedded systems need to be resistant again both software attacks (*e.g.,* network vulnerabilities) *and* physical attacks (*e.g.,* hardware-induced faults).

These differences are not trivial, and have created a particularly challenging environment for security researchers. In fact, there is currently no system capable of analyzing embedded systems in the general case, neither statically nor dynamically. Similarly, the physical attack vector is not one that most software engineers are accustom to defending against, and thus most of the embedded systems today are vulnerable to even simple "glitching" attacks.

If there is any hope for defenders to get the upper hand, as they have with malware, we must create efficient analysis systems *and* introduce defenses that fundamentally mitigate the most important attacks. Therefore, my research, and this body of work, is my attempt to address these critical needs and my hope is that it will help guide future research so that we can create a future where attackers are the ones playing an unbalanced game that is stacked against them.

Chapter 3 presents a hardware-introspection tool capable of performing low-artifact analysis of any software using hardware introspection (*i.e.,* the hardware is instrumented to observe what the software is doing, without disturbing the actual software itself). This low-artifact, difficult-to-detect introspection is particularly valuable for hard-to-analyze embedded systems, which may not have debugging enabled or may be prohibitively difficult to emulate.

Chapter 4 presents a method for automatically modeling and emulating hardware peripherals to support emulation. The goal of this work was to enable scalable, large-scale analysis of embedded firmware through virtualization (*i.e.,* similar to what was done with malware).

Chapter 5 provides valuable insights into the world of glitching attacks, where fact and fiction have become increasingly conflated — attackers have made some outlandish claims and numerous defenses have been proposed without any rigorous analysis into their efficacy. Thus, we performed an array of experiments to separate theory from practice. In the process, we created an open-source tool capable of automatically instrumenting any software, at compile time, with software-only glitching defenses that we show to be to highly effective at detecting glitching attacks in practice (*i.e.,* over a 99% detection rate).

Chapter 3

Low-Artifact Analysis Using Hardware Introspection

3.1 Introduction

With the rapid advancement of malware, the capabilities of existing analysis techniques have become obsolete. Tools that exist within the operating system or hypervisor are prone to creating artifacts that are visible to the malicious code. Malware authors can leverage these artifacts to conceal their true intentions by halting execution or potentially subverting the analysis technique all together. Even with clever designs, there remains no proven technique for developing low-artifact software-based analysis tools. Moreover, recent work by Kirat *et al.* [16] showed that at least 5% of the malware analyzed in their study employed anti-analysis techniques that successfully evade most existing analysis tools. Subsequently, numerous systems have been developed that attempt to detect and analyze these *environment-aware* malware (*e.g.,* [50, 51]). Chen *et al.* [52] even provide a taxonomy of anti-analysis techniques and mitigations commonly employed. However, no bullet-proof solutions exist, and most existing solutions require continuous updating as they rely on emulation frameworks.

To address these problems we present LO-PHI (Low-Observable Physical Host Instrumentation), a novel system capable of analyzing software executing on commercial-off-the-shelf (COTS) bare-metal machines, without the need for any additional software on the machines. LO-PHI permits accurate monitoring and analysis of live-running physical hosts in real-time, with a minimal addition of "plug-and-play" components to an otherwise unmodified system under test (SUT). We have taken a two-pronged approach that is capable of instrumenting machines with actual hardware, to minimize artifacts, or with traditional software-based techniques utilizing hardware virtualization, to maximize scale. This permits the tradeoff between transparency, scale, and cost when appropriate as well as the potential for parallel analysis. Our architecture uses physical hardware and software-based sensors to monitor a SUT's memory, network, and disk activity as well as actuate its keyboard, mouse, and power. The raw data collected from our sensors is then processed with modified open source tools, *i.e.,* Volatility [53] and Sleuthkit [54], to bridge the *semantic gap*, *i.e.,* convert raw data into human-readable, semantically rich, output. Because LO-PHI is designed to collect raw low-level data, it is both operating system and file system agnostic. Our framework can be easily extended to support new or legacy operating systems and file systems as long as the hardware tap points are suitable for data acquisition.

LO-PHI also necessitated the development of novel introspection techniques. While numerous techniques exist for memory acquisition [55–57] of bare-metal systems, passively monitoring disk activity has only recently begun to be explored [58]. In this work we present a hardware sensor capable of passively sniffing the disk activity of a live machine while introducing minimal artifacts. Subsequently, we have also developed the required modules for parsing and reconstructing the underlying serial advanced technology attachment (SATA) protocol. All of the source code for LO-PHI is available under the

Berkeley Software Distribution (BSD) license at `http://github.com/mit-ll/LO-PHI`.

While the potential applications for LO-PHI are vast, we focus on our ability to perform automated malware analysis on physical machines, and demonstrate its usefulness by showcasing the ability to analyze classes of malware that trivially evade existing dynamic analysis techniques. We first briefly summarize the current state of dynamic malware analysis to better highlight our contributions in *Section* 3.2. Next, we describe the design and implementation of our system, including hardware sensors for memory and disk capture as well as actuators for controlling and reverting a system under test (SUT) in *Section* 3.3. We then attempt to quantify the exposed hardware artifacts of our system *Section* 3.4 and some of our inherent limitations in *Section* 3.5. We discuss the design of our automated binary analysis framework in *Section* 3.6 and present the findings from our analysis of various malware samples in *Section* 3.7. Finally, we compare LO-PHI to other related works in *Section* 6.1 and highlight areas that we feel are rich for future work in *Section* 3.8.

In summary we claim to make the following contributions to the field of dynamic analysis:

- Deployed and tested an extremely low-artifact, hardware-based, dynamic analysis environment capable of analyzing malware that avoids traditional software-based techniques

- Developed hardware capable of introspecting the communication between SATA devices as well as asynchronous memory acquisition

- Wrote a module capable of reconstructing SATA frames into high-level disk sector operations

- Modified open-source forensics tools to reconstruct file system and operating system states

- Constructed a framework, and accompanying infrastructure, for automating analysis of binaries on both physical and virtual machines
- Demonstrated the scalability of our system to execute and analyze thousands of samples with comparable fidelity to traditional VM-based solutions

3.2 Background and Threat Model

In this section, we introduce the vocabulary and basic concepts surrounding the LO-PHI system as well as the scope of our threat model.

Stealthy Malware Recent malware detection and analysis tools rely on virtualization, emulation, and debugging tools. Unfortunately, these techniques are becoming obsolete with the growing interest in *stealthy malware*. Malware is stealthy if it makes an effort to hide its true behavior. This stealth can emerge in several ways.

First, malware can simply remain inactive in the presence of an analysis tool. Such malware will use a series of platform-specific tests to determine if certain tools are in use. If no tools are found, then the malware executes its malicious payload.

Second, malware may abort its host's execution. For example, a sample may attempt to execute an esoteric instruction that is not properly emulated by the tool being used. In this case, attempting to emulate the instruction may lead to raising an unhandled exception, crashing the program.

Third, malware may simply disable defenses or tools altogether. For instance, OllyDbg would crash when attempting to emulate `printf` calls with a large number of '%s' tokens. This type of malware may also infect kernel space and then disable defenses by abusing its elevated privilege level.

Artifacts As mentioned above, stealthy malware evades detection by concluding whether an analysis tool is being used to watch its execution. This means that there must be some piece of evidence available to the malware that it uses to make this determination, commonly known as the *observer effect*. This may be anything from execution time (*e.g.,* debuggers make programs run more slowly) to I/O device names (*e.g.,* if a device has a name with 'VMWare' in it), to emulation idiosyncrasies (*e.g.,* QEMU fails to set certain flags when executing obscure corner-case instructions). We refer to these bits of evidence as *artifacts*. LO-PHI seeks to make instrumentation and measurement of malware more transparent by reducing or eliminating the presence of these artifacts.

Malware Analysis These stealth techniques have necessitated the development of increasingly sophisticated techniques to analyze them. For benign or non-stealthy binaries, numerous debuggers exist such as OllyDbg, Immunity, and gdb. These debuggers can be trivially detected in most cases (*e.g.,* by using the `isDebuggerPresent()` function). These anti-analysis techniques led researchers to develop more transparent, security-focused analysis frameworks using virtual machines, which generally work by hooking system calls to provide an execution trace which can then be analyzed [4, 5, 59–62]. System call interposition has its own inherent problems [63] (*e.g.,* performance overhead), which led many researchers to decouple their analysis code even further from the execution path. Virtual-machine introspection (VMI) peeks at system state without any direct interaction with the control flow of the program, thus mitigating much of the performance overhead. VMI has prevailed as the dominant low-artifact technique and has been used by numerous malware analysis systems [64–70]. Jain *et al.* [71] provide an excellent overview of this area of research. However, introspection techniques have very limited access to the semantic meaning of the data that they are acquiring, which led to the development of numerous techniques for bridging this *semantic gap* in both memory [53, 67, 72–74] and disk [54, 58, 75] accesses. All VM-based techniques thus far have nevertheless been shown to reveal some detectable artifacts [52, 76–78] that could still be used to subvert analysis [78, 79].

Because the techniques used to bridge the semantic gap rely only on raw data and are in no way tied to the method of acquisition, newer techniques further decouple the analysis code from the SUT by moving the analysis portion into system management mode (SMM) mode, an isolated execution mode available on x86 processors, [57,80–82] or onto a separate processor altogether [83–87]. Similarly, our work fueled the development of an array of methods for acquiring a system's memory and disk state, while introducing even fewer artifacts. In decreasing order of artifacts, the most popular techniques for acquiring memory use either specialized software [88,89] or hardware to exploit direct memory access (DMA) over FireWire [55,90] or using a peripheral component interconnect (PCI) card [56,91]. Molina *et al.* [86] used a PCI card to obtain an out-of-band method for accessing the hard disk. While a few techniques have been proposed to defeat introspection and semantic-gap based approaches [92], they tend to be very fragile in practice and are not likely to be widely deployed. With LO-PHI, we hope to help bridge the gap between these low-artifact data acquisition methods and the semantically-rich emulated analysis frameworks to provide similar output with far less overhead.

Threat Model While LO-PHI has an almost-complete view of the SUT, and introduces very few artifacts, we still make a few necessary assumptions about the malware that we are analyzing. Specifically, we assume that the malware can interact with the SUT without restriction, but that any malicious modifications made to the system will be visible either in main memory or on the disk drive. Malware capable of infecting peripherals or other onboard chips are currently out of scope for our system. Similarly, we assume that the malware in question is not actively trying to thwart semantic-gap reconstruction or avoid our particular hardware through signature-based means. Finally, we assume that our instrumentation was in place before the malicious code was executed, to ensure that our exposed artifacts cannot be fingerprinted. That is, the malware has no chance to analyze the SUT without LO-PHI in place, and thus cannot distinguish our analysis system from a system without our instrumentation. Otherwise, we assume that malware may employ any exploitation, anti-analysis, or anti-debugging techniques. More precisely, LO-PHI is specifically designed to detect highly-sophisticated stealthy malware.

3.3 System Implementation

LO-PHI leverages various sensors, actuators, and software analysis tools combined into a simple and scalable framework. For the purpose of our framework we generally define a *sensor* as any data collection component (*e.g.,* memory, disk, or network) and an *actuator* as any component which provides inputs for the system (*e.g.,* power, keyboard, or mouse). Our architecture allows for simple one-off experiments on a single target SUT as well as much larger analyses running in parallel on multiple SUTs. The hardware sensors support high-speed, low-artifact collection of various data from a SUT, such as memory or disk activity. Similarly, we employ hardware actuators, which automatically drive a SUT to set up and run experiments, as well as clean up afterwards. The software analysis tools run on a separate analysis machine, capable of aggregating and analyzing the collected data in real time.

In addition to the hardware, LO-PHI also supports traditional virtual-machine introspection using software-based sensors and actuators within our framework. While the major contribution of this paper is our hardware instrumentation, much of our framework's power stems from its duality. We have implemented all of our capabilities in both virtual and physical environments within the same abstracted software interface written in Python. This permits the development of tools that will seamlessly work in either environment, physical or virtual, depending on their instantiation. For example, analysis scripts need only implement high-level functionality (*e.g.,* `memory_read()`, `power_on()`, `disk_revert()`), which our framework will execute appropriately for the given machine type. Similarly, we devised a scripting language for keyboard and mouse actions, along with an appropriate parser for each instantiation.

3.3.1 Physical Instrumentation

While much work has been done in instrumenting and introspecting virtual machines, we are only aware of a few systems [15, 16, 93] that have had the goal of bare-metal instrumentation. We feel that the lack of existing solutions is likely due to a lack of motivation and the high barrier of entry, *i.e.,* it is more costly in both human effort and resources to instrument physical machines. In this work, we hope to highlight the usefulness of these techniques and advance the state-of-the-art in malware analysis.

One of our major design goals was to create tools that could be utilized on a wide range of existing commercial-off-the-shelf (COTS) hardware with minimal modification. While implementations with fewer artifacts are possible with specialized hardware (*e.g.,* memory interposers) or modifications of existing hardware (*e.g.,* firmware modifications), developing more robust sensors permits a wider range of potential analyses and helps future-proof our techniques. For these reasons, we chose to focus on two major sources of data for our physical sensors: main memory and disk activity. In particular, we employed Xilinx field-programmable gate array (FPGA) boards to interface with both the peripheral component interconnect express (PCIe) interface, for memory acquisition, and the Serial ATA (SATA) interface, for disk introspection. All of our sensors and actuators communicate over gigabit Ethernet via the user datagram protocol (UDP).

Memory (Physical)

Our memory sensor is implemented on a Xilinx ML507 development board. This particular board has single-lane PCIe connector, which we utilize as an endpoint to read the SUT's memory using direct memory access (DMA). We instantiate the card as a bus master by enabling the *Bus Master Enable* bit in the card's configuration register. While each peripheral is designated a distinct memory region for DMA, there were traditionally no enforcement mechanisms to stop a peripheral from reading and writing arbitrary memory locations. This method has been widely studied [56, 57, 94] and exploited [55, 94–99] over the years. Subsequently, PCIe can achieve very rapid polling rates, making it an ideal candidate for reliable memory acquisition.

Disk (Physical)

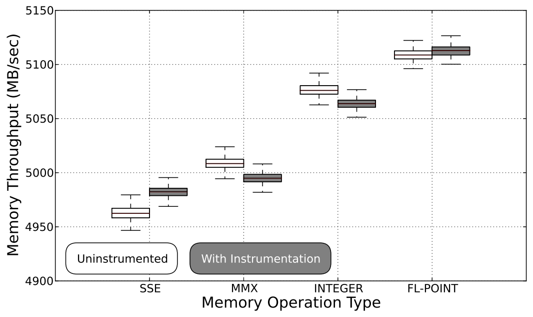
We also employ the ML507 board for our disk analysis by using its two onboard SATA connectors. An Intelliprop SATA bridge core (Part number: IPP-SA110A-BR) provides the ability to passively monitor, and potentially manipulate, all of the traffic over the SATA interface between the host and device. To receive this data on our remote analysis host, we implemented the logic to package the SATA frames into UDP packets and send them over the gigabit Ethernet connection. This proved a difficult engineering feat, as the data rates of SATA exceed the capacity of our gigabit Ethernet link, and thus necessitated numerous data-flow integrity guarantees. This interface is completely passive and is essentially invisible to the SUT, aside from the occasional throttling of frames.

Actuation (Physical)

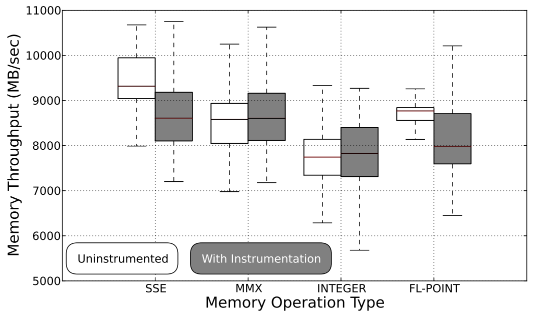
For many of our intended applications, it is convenient, and sometimes necessary, to actuate the SUT from our analysis scripts. For this purpose, we employ the Arduino Leonardo, which is driven by a ATmega32u4. It has numerous general-purpose input/ouput (GPIO) pins, as well as the ability easily emulate a keyboard and mouse through the universal serial bus (USB) interface. We use an external power source for the Arduino to permit functions such as powering on the SUT through the GPIO pins attached to the SUT's motherboard. Integrating the Leonardo into our software framework was relatively straightforward, given the simplistic development environment provided for Arduino platforms. That said, correctly emulating mouse movements required some additional effort. However, once implemented, these mouse movements provide LO-PHI with the capability to move the mouse as a human user would (*e.g.,* continuously move the mouse) and also click buttons presented by the software. While numerous commercial solutions already exist for some of this instrumentation (*e.g.,* Intelligent Platform Management Interface, Active Management Technology, Dell Remote Access Card), we wanted to ensure that LO-PHI would be usable on the widest-range of systems possible, and thus opted for the lower-level interfaces, specifically USB.

Infrastructure (Physical)

While our sensors and actuators achieve all of our low-level requirements for instrumentation, numerous actuation functions (*e.g.,* reverting the disk or checking the OS's boot status) required the development of specialized infrastructure. While reverting the disk for a virtual machine is as easy as overwriting a file, reverting a physical machine quickly becomes much more involved. Our requirement of unmodified hardware eliminates options such as network booting or specialized drives, which would also produce



(a) Physical machine (Polling at 14MB/sec)



(b) Virtual machine (Polling at 160MB/sec)

Figure 3.1: Average memory throughput comparison as reported by RAMSpeed, with and without instrumentation on both physical and virtual machines. (500 samples for each box plot)

significant detectable artifacts.

To achieve the desired outcome, we implemented our own preboot execute environment (PXE) server as well as an accompanying trivial file transfer protocol (TFTP) server, dynamic host configuration protocol (DHCP) server and domain name service (DNS) server. This enables us to temporarily permit a given media access control (MAC) address to boot a Clonezilla [100] instance, which restores the disk to a previously saved state. By doing this, we make our system more flexible and scalable as the hardware is no longer tied to a particular operating system or installation configuration. Similarly, by hosting our own DNS and DHCP server, we are able to simplify our scripts and create a richer atmosphere for malware analysis (*e.g.,* we can trivially lookup the IP of a given machine since our infrastructure assigns it). We also use gigabit Netgear GS108T switches, one per physical machine, and utilize virtual local-area networks (VLANs) to ensure that our control and sensor traffic do not interfere with each other.

3.3.2 Virtual Instrumentation

Since a great deal of work has already been done using virtual-machine introspection (VMI), we choose to leverage existing capabilities when possible by simply incorporating them into our software framework with minimal amounts of glue code. Because of our desire to use existing solutions, and source code availability, we chose to use open-source hypervisor implementations, namely QEMU/KVM [101, 102].

Memory (Virtual)

For live memory acquisition, we use techniques similar to those employed by *LibVMI* [103]. We obtain access to the guest's physical memory by means of a UNIX socket. This socket then permits arbitrary memory read and write commands, which perform the appropriate action on the guest's memory using `cpu_physical_memory_map` and `cpu_physical_memory_unmap`.

Disk (Virtual)

To incorporate disk introspection into our virtual environment, we inserted hooks into the QEMU block driver. These hooks intercept all disk operations and copy the relevant data to a separate thread, which then exports them over a UNIX socket to a subscription sever. By spawning a new thread for every access, we should have negligible impact on the system performance, especially if the host system has underutilized resources. The server then allows our clients to connect and subscribe to the disk activity of any guest. We also ensured that our implementation works properly with copy-on-write disks, greatly increasing performance when an experiment requires frequently resetting the disk state.

Actuation (Virtual)

For actuation, we leverage libvirt [104], an open-source tool for interacting with hypervisors. Again, mouse movement proved to be less straightforward, and necessitated the development of a custom virtual network computing (VNC) client.

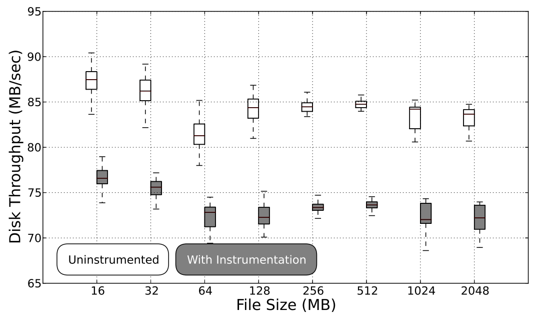
3.4 Artifacts

While hardware-level introspection provides numerous desirable security guarantees that are not available for software-based solutions (*e.g.,* hardware segregation of analysis code), it is still critically important to introduce minimal artifacts and, in the ideal case, none at all. We emphasize that the artifacts produced by LO-PHI are likely unusable by malware for subversion, because the malware would lack a baseline for comparison. Minimizing artifacts not only improves the fidelity of our data and the performance of our system, but also reduces the number of "tells" on a SUT that can be used to evade or hinder our analysis. Nevertheless, we enumerate the artifacts introduced by our instrumentation and attempt to address any shortcomings in both virtual and physical machines. All of our experiments were run using our infrastructure to automate the execution of our benchmarking applications on both physical and virtual machines both with and without instrumentation. In our performance experiments, the physical SUT was a Dell T7500 equipped with an 6-core Xeon X5670 and 2GB of RAM and a WD3200AAKX disk drive, and our virtual machine was instantiated on a Dell T7600 equipped with dual 8-core Xeon E5-2665 processors and 68GB of RAM, with the VM itself allocated one core, 1GB of RAM and a 10GB hard disk.

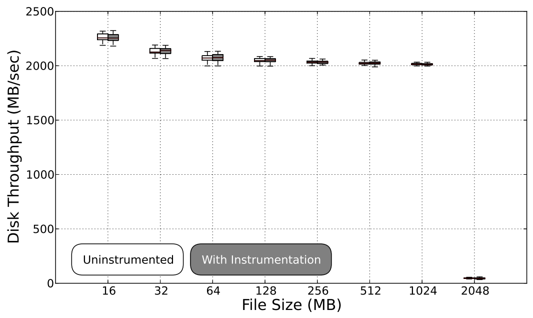
3.4.1 Memory Artifacts

In both of our memory-introspection techniques, *i.e.,* physical and virtual, there is likely to be a performance impact on the SUT, as we are accessing a shared resource (main memory). We attempted to quantify this performance impact by utilizing *RAMSpeed*, a popular RAM benchmarking application. We ran RAMSpeed on the same system with and without our instrumentation. In each case, we conducted four experiments designed to stress each of the INT, MMX, SSE, and FL-POINT instruction sets. Each of these experiments consists of 500 sub-experiments, which evaluate and average the performance of copy, scale, sum, and triad operations. To ensure that the memory reads for our instrumentation were not being cached, we had our sensors continuously read the entire memory space, which should have also introduced the largest performance impact on the system. The memory polling rates were dictated by the hardware and our particular implementation.

At first glance, *Figure* 3.1 may seem to indicate that our instrumentation has a discernible effect on the system; however, the deviation from the uninstrumented median is only 0.4% in the worst case (SSE in *Figure* 3.1*a*). Despite our best efforts to create a controlled experiment, *i.e.,* running RAMSpeed on a fresh install of Windows with no other processes, we were unable to to definitively attribute any deviation to our polling of memory. While our memory instrumentation certainly has some impact on the system, the rates at which we are polling memory do not appear to be frequent enough to predictably degrade performance. This result appears to indicate that systems like ours could poll at significantly higher rates while still remaining undetectable. For example, PCIe implementations can achieve DMA read speeds of 3 GB/sec [105], which could permit a new class entirely of introspection capabilities. To this end, we have successfully achieved rates as fast as 60 MB/sec using SLOTSCREAMER [106]; however, the imple-



(a) File writes



(b) File reads

Figure 3.2: File system throughput comparison as reported by IOZone on Windows XP, with and without instrumentation on a physical machine. (50 samples for each box plot)

mentation is not yet stable enough to incorporate into our framework. Nevertheless, in order to detect any deviation in performance, the software being analyzed would need to have the ability to baseline our system, which is not feasible in our experiments.

While performance concerns are a universal problem with instrumentation, adding hardware to a physical configuration has numerous additional artifacts that must also be addressed. To utilize PCIe, we must enumerate our card on the bus, which means that the BIOS and operating system are able to see our hardware. This inevitably reveals our presence on the machine; nevertheless, mitigations do exist (*e.g.,* masquerading as a different device). To avoid detection, our card could trivially use a different hardware identifier every time to avoid signature-based detection. Even with a masked hardware identifier however, Stewin *et al.* [107] demonstrated that all of these DMA-based approaches will reveal some artifacts that are exposed in the CPU performance counters. Similar techniques could be employed by malware authors in both physical and virtual environments to detect the presence of a polling-based memory acquisition system such as ours. These anti-analysis techniques could necessitate the need for more sophisticated acquisition techniques, some of which are proposed in *Section* 3.8.

3.4.2 Disk Artifacts

To quantify the performance impact of our disk instrumentation, we similarly employed a popular disk benchmarking utility, *IOZone* [108]. While IOZone's primary purpose is to benchmark the higher-level filesystem, any performance impacts on disk throughput should nonetheless be visible to the tool. We used the same setup as the previous memory experiments and ran IOZone 50 times for each case, *i.e.,* with and without our instrumentation, monitoring the read and write throughput with a record size of 16MB and file sizes ranging from 16MB to 2GB (the total amount of RAM on SUT).

Our hardware should only be visible when we intentionally delay SATA frames to meet the constraints of our gigabit Ethernet link. We designed the system with this delay to minimize our packet loss since UDP does not guarantee delivery of packets. In practice, we rarely observed the system cross this threshold; however, IOZone is explicitly made to stress the limits of a file system. For smaller files, caching masks most of our impact as these cache hits limit the accesses that actually reach the disk. The caching effect is more prevalent when looking at the raw data rates (*e.g.,* the median uninstrumented read rate was 2.2GB/sec for the 16MB experiment and 46.2MB/sec for 2GB case).

The discrepancies between the read and write distributions are attributed to the underlying New Technology File System (NTFS) and optimizations in the hard drive firmware. *Figure* 3.2*b* shows that our instrumentation is essentially indistinguishable from the base case when reading, the worst case being a degradation of 3.7% for 2GB files. With writes however, where caching offers no benefit, the effects of our instrumentation are clearly visible, with a maximum performance degradation of 14.5%. Under typical operating conditions, throughputs that reveal our degradation are quite rare. In these experiments, the UDP data rates observed from our sensor averaged 2.4MB/sec with burst speeds reaching as high as 82.5MB/sec, which directly coincide with the rates observed in *Figure* 3.2*a*, confirming that we are only visible when throttling SATA to meet the constraints of the Ethernet connection.

In the case of virtual machines, we would expect to have no detectable artifacts on a properly provisioned host aside from the presence of a kernel virtual machine (KVM). This is because our instrumentation adds very little code into the execution path for disk accesses, and uses threading to exploit the numerous cores on our system. More precisely, our instrumentation only adds a memory copy operation of the data buffer, which is then passed to a thread to be exported. Our experimental results confirmed this hypothesis as we were unable to identify any consistent artifacts in our IOZone tests.

3.5 Limitations

All of our techniques have some inherent limitations. We attempt to enumerate the most prominent of those below.

Input/Output Memory Management Unit (IOMMU) Newer chipsets are equipped with IOMMUs, which, when properly configured to disable DMA from peripherals, would render our current memory acquisition approach ineffective. While this limits us from instrumenting arbitrary systems, it does not thwart our approach in the analysis case, *i.e.,* where we have complete control of the system that we are instrumenting, because we could simply disable this functionality or purchase chipsets that do not contain IOMMUs. In the long term however, we will likely have to migrate our techniques to employ a different memory acquisition method.

Asynchronous Memory Access Wang *et al.* [109] provide a good analysis of the inherent limitations of polling-based systems for malware detection and potential evasion techniques. However, memory polling may also create issues with *smearing* and caching as well. Smearing occurs when the state of memory of a SUT changes during acquisition, resulting in an imperfect memory capture over a time window rather than a specific instant in time. The memory contents of live SUTs are very dynamic, so smearing is likely to occur. Nevertheless, we have only rarely had this effect cause problems in practice, and faster polling rates would help minimize the smearing effect. Similarly, there may be rare cases where data never leaves a cache; however, this can be easily mitigated with cache coherency.

Filesystem Caching For disk monitoring, OS-level disk caching may cause our disk sensor to miss SATA frames that hit the cache and are overwritten before being flushed to disk. While this effect is likely to be minor during continuous disk monitoring, it is conceivable that malware could drop a file, execute it, and delete it before the cache is ever flushed to disk, completely evading detection. However, we do not see this as a major issue, as attempts of persistence will eventually have to write to disk. Additionally, the effects of the malware would also likely be detectable in memory.

No Internet Access Our experiments also had a few limitations that were beyond our control. For example, the network policies within our organization currently forbid us from running these malware samples on the live Internet. Similar studies have concluded that most malware from the wild will appear to do nothing, aside from network activity, without the presence of its command-and-control infrastructure from which to retrieve a payload. Because of this, we do not present our results as representative of the presence of VM-aware malware in the wild, but instead highlight our capabilities and the ability to detect particularly sophisticated payloads once they are executed.

3.6 Experimental Framework

To facilitate experimentation, we built a scalable infrastructure capable of running arbitrary binaries on either a physical or virtual machine with a specified operating system. Our software infrastructure consists of a **master** which accepts job submissions and delegates them to an appropriate **controller**. A given **controller** is initialized with a set of **machines**, both physical and virtual SUTs, that serve as its worker pool. Upon a job submission, the **controller** first downloads the script, which describes the actions to perform on the SUT, and submits the job to a scheduler. This scheduler then waits for a **machine** of the appropriate type, *i.e.,* physical or virtual, to become available in the pool, allocates it to the analysis, and runs the requested routines. All of our malware samples, analyses, and results were stored in a MongoDB database. Samples were submitted using a custom FTP server and a command line tool that interfaced with the **master** to instantiate a given analysis script, which are stored on the **master** and dynamically sent to the **controller**.

Because of the duality of our framework we were able to write one simple script (see *Figure* 3.3) that will: 1) reset our machine to a clean state, 2) take a memory image before and after execution, 3) attempt to click any graphical buttons, 4) capture screenshots, and 5) capture all disk and network activity throughout the execution. To download and execute an arbitrary binary (*Figure* 3.3, line 12), our implementation uses hotkeys to open a command line interface, executes a recursive file-transfer protocol (FTP) download to retrieve the files to be analyzed, and then runs a batch file to execute the binary. From this data, we reconstruct the changes in system memory, in addition to a complete capture of disk and network activity generated by the binary. To identify any graphical buttons that the malware may present, we use the Volatility "windows" module to identify all visible windows that have an atom class of `0xc061` or an atom

```
# Reset our disk using PXE
machine.machine_reset()
machine.power_on()
# Wait for OS to appear on network
while not machine.network_get_status():
    time.sleep(1)
# Allow time for OS to continue loading
time.sleep(OS_BOOT_WAIT)
# Start disk capture
disk_tap.start()
# Send key presses to download binary
machine.keypress_send(ftp_script)
# Dump memory (clean)
machine.memory_dump(memory_file_clean)
# Start collection network traffic
network_tap.start()
# Get a list of current visible buttons
button_clicker.update_buttons()
# Start our binary and click any buttons
machine.keypress_send('SPECIAL:RETURN')
# Move our mouse to imitate a human
machine.mouse_wiggle(True)
# Allow binary to execute (Initial)
time.sleep(MALWARE_START_TIME)
# Dump memory (interim)
machine.memory_dump(memory_file_interim)
# Take a screenshot (Before clicking buttons)
machine.screenshot(screenshot_one)
# Click any new buttons that appeared
button_clicker.click_buttons(new_only=True)
# Allow binary to execute (3 min total)
time.sleep(MALWARE_EXECUTION_TIME-elapsed_time)
# Take a final screenshot
machine.screenshot(screenshot_two)
# Dump memory (Dirty)
machine.memory_dump(memory_file_dirty)
# Shutdown Machine
machine.power_shutdown()
```

Figure 3.3: Python script for running a malware sample and collecting the appropriate raw data for analysis.

superclass of 0xc017, which indicate a button, and then use our actuator to move the mouse to the appropriate location and click it. Our analysis framework also attempts to remove any typical analysis-based artifacts by using a random file name and continuously moving the mouse during the execution of the binary. Similarly, when possible, *i.e.,* the system is not hung, we also properly shutdown the system at the end of the analysis to force any cached disk activity to be flushed.

In our analysis setup, both the physical and virtual environments had a 10 GB partition for the operating system and 1 GB of volatile memory. The operating system

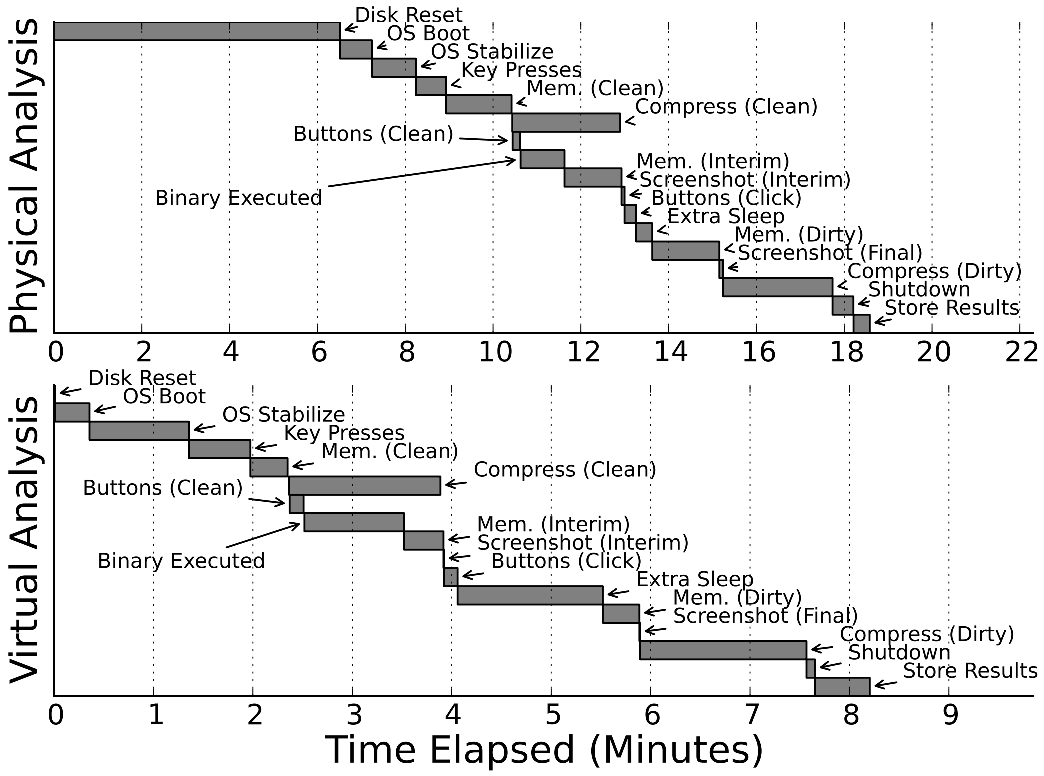
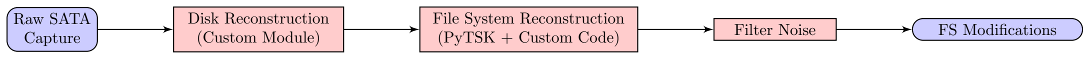


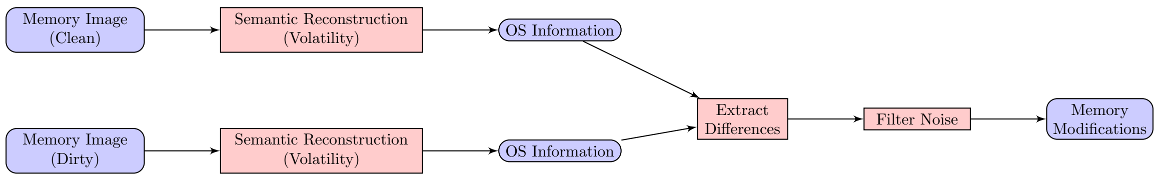
Figure 3.4: Time spent in each step of binary analysis. Both environments were booting a 10 GB Windows 7 (64-bit) hibernate image and were running on a system with 1 GB of volatile memory.

was placed into a "hibernate" state to minimize the variance between executions and also reduce the time required to boot the system. To minimize the space requirements of our system, we compress our memory images before storing them in our databases. While this adds a significant amount of time to our analysis (approximately 2 minutes), it significantly reduces the storage requirement. Finally, the virtual machine's networks were logically divided to ensure that samples did not interfere with each other, and the physical environment consisted of only one machine.

The respective runtimes for each portion of our analysis can be seen in *Figure* 3.4. We ensured that every binary executed for at least 3 minutes before retrieving our final memory image and resetting the system. Screenshots were obtained using Volatility's


Raw SATA Capture
Disk Reconstruction (Custom Module)
File System Reconstruction (PyTSK + Custom Code)
Filter Noise
FS Modifications

(a) Disk Reconstruction



(b) Memory Reconstruction

Figure 3.5: Binary analysis workflow. (Rounded nodes represent data and rectangles represent data manipulation.)

screenshot module on physical machines and were extracted from the captured memory images. Note that most of the time taken in the physical case is due to our resetting of the system state using Clonezilla, waiting for the system to boot, and memory acquisition. The resetting and boot process could be decreased significantly by writing a custom PXE loader, or completely mitigated by implementing copy-on-write into our FPGA. Similarly, the memory acquisition times could be more comparable to the virtual case, if not faster, by optimizing our PCIe implementation. Finally, *system snapshots* could reduce the time spent setting up the environment to mere seconds. While snapshots are trivial with virtual machines, it is still an open problem for physical machines.

We note that LO-PHI may miss any temporal memory modifications made by the binary between our clean and dirty memory images. To analyze the transient behavior of a binary, LO-PHI could be used to continuously poll the systems memory during execution. However, while this has the potential to produce a lot more fidelity, we do not feel that our current polling rates are fast enough to warrant the tradeoff between the produced DMA artifacts and usefulness of the output. We hope to explore this area of research in more detail in the future as we improve our memory capture capabilities.

3.7 Evaluation and Analysis

In this section, we explain our methodology for semantic gap reconstruction (*Section* 3.7.1) and demonstrate the practicality of LO-PHI with three targeted experiments. The experiments were constructed to demonstrate the following:

- The ability of LO-PHI to detect the behaviors elicited by real malware, confirmed with ground truth (*Section* 3.7.3)
- The ability to scale and extract meaningful results from unknown malware samples (*Section* 3.7.4)
- The ability to analyze malware samples that employ anti-analysis and evasion techniques (*Section* 3.7.5)

For each binary, we determine the system changes that occurred during execution by forensically comparing the resulting *clean* and *dirty* states. Each such pair of datasets contains a clean and a dirty raw memory snapshot respectively as well as a log of raw disk and network activity that occurred between clean and dirty states. We exclude the network trace analysis from much of our discussion since it is a well-known technique and not the focus of our work. Our analysis of a binary's execution involves four steps: 1) bridging the semantic gap for both the clean and dirty states, 2) computing the delta between the two states, 3) filtering out actions that are not attributed to the binary, and 4) comparing the delta for physical execution and virtual execution to determine if the sample employs VM-detection techniques (if applicable). The process taken for each binary is illustrated *Figure* 3.5. When appropriate, we also compare our results to those produce by Anubis [5] and Cuckoo Sandbox [3].

Offset	**Name**	**PID**	**PPID**
0x86292438	AcroRd32.exe	1340	1048
0x86458818	AcroRd32.exe	1048	1008
0x86282be0	AdobeARM.exe	1480	1048
0x864562a0	*$$_rk_sketchy_server.exe*	1044	1008

(a) New Processes (pslist)

PID	**Port**	**Protocol**	**Address**
1048	1038	UDP	127.0.0.1
1044	*21*	*TCP*	*0.0.0.0*

(b) New Sockets (sockets)

Selector	**Base**	**Limit**	**Type**	**DPL**	**Gr**	**Pr**
0x320	0x8003b6da	0x00000000	CallGate32	3	-	P

(c) GDT Hooks (gdt)

Name	**Base**	**Size**	**File**
hookssdt.sys	0xf7c5b000	0x1000	C: \...\lophi\hookssdt.sys

(d) Loaded Kernel Models (modscan)

Table	**Entry Index**	**Address**	**Name**	**Module**
0	0x0000f7	0xf7c5b406	NtSetValueKey	hookssdt.sys
0	0x0000ad	0xf7c5b44c	NtQuerySystemInformation	hookssdt.sys
0	0x000091	0xf7c5b554	NtQueryDirectoryFile	hookssdt.sys

(e) SSDT Hooks (ssdt)

Created Filename
/.../lophi/$$_rk_sketchy_server.exe
/.../lophi/hookssdt.sys
/.../lophi/sample_0742475e94904c41de1397af5c53dff8e.exe

(f) Disk Event Log (81 Entries Truncated)

Figure 3.6: Post-filtered semantic output from rootkit experiment (*Section* 3.7.3).

3.7.1 Semantic Gap Reconstruction

As previously mentioned, before any analysis can be conducted, we must first bridge the semantic gap, *i.e.,* translate our memory snapshots and SATA captures, which contain low-level, raw, data into high-level, semantically-rich, information.

Memory

To extract operating-system-level modifications from our memory captures, we run a number of Volatility plugins on both clean and dirty memory snapshots to parse kernel structures and other objects. Some of the general purpose plugins include *psscan*, *ldrmodules*, *modscan*, and *sockets*, which extract the running processes, loaded dlls, kernel drivers, and open sockets resident in memory. Similarly, we also run more malware-focused plugins such as *idt*, *gdt*, *ssdt*, *svcscan*, and *callbacks* which examine kernel descriptor tables, registered services, and kernel callbacks.

Disk

The first step in our disk analysis is to first convert the raw capture of the SATA activity into a 4-tuple containing the disk operation (*e.g.,* READ or WRITE), starting sector, total number of sectors, and data. Our physical drives, as with most modern drives, used an optimization in the SATA specification known as Native Command Queuing (NCQ) [110]. NCQ reorders SATA Frame Information Structure (FIS) requests to achieve better performance by reducing extraneous head movement and then asynchronously replies based on the optimal path. Thus, to reconstruct the disk activity, our SATA reconstruction module must faithfully model the SATA protocol in order to track and restore the semantic ordering of FIS packets before translating them to disk operations. Upon reconstructing the disk operations, these read/write transactions are then translated into disk events (*e.g.,* filesystem operations, Master Boot Record modification, slack space modification) using our analysis code which is built upon Sleuthkit and PyTSK [111]. Since Sleuthkit only operates on static disk images, our module required numerous modifications to keep system state while processing a stream of disk operations. Intuitively, we build a model of our SUT's disk drive and step through each read and write transaction, updating the state at each iteration and reporting appropriately. This entire process is visualized in $Figure$ 3.5a. Unlike previous work [58], which was designed for NTFS, our approach is generalizable to any filesystem supported by Sleuthkit. A sample output from creating the file *LO-PHI.txt* on the desktop can be seen below:

MFT modification (Sector: 6321319)					
Filename	/WINDOWS/.../drivetable.txt$\rightarrow$/.../Desktop/New Text Document.txt				
Allocated	0 $\rightarrow$ 1	**Unallocated**	1 $\rightarrow$ 0	**Size**	132 $\rightarrow$ 0
Modified	2014-11-07 20:07:06 (1406250) $\rightarrow$ 2015-02-19 15:47:17 (3281250)				
Accessed	2014-11-07 20:07:06 (1406250) $\rightarrow$ 2015-02-19 15:47:17 (3281250)				
Changed	2014-11-07 20:07:06 (1406250) $\rightarrow$ 2015-02-19 15:47:17 (3281250)				
Created	2014-11-07 20:07:06 (1406250) $\rightarrow$ 2015-02-19 15:47:17 (3281250)				

...

MFT modification (Sector: 6321319)	
Filename	/.../Desktop/New Text Document.txt $\rightarrow$/.../Desktop/LO-PHI.txt
Changed	2015-02-19 15:47:17 (3281250) $\rightarrow$ 2015-02-19 15:47:25 (3437500)

Note that we can infer from this output that the filesystem reused an old MFT entry for *drivetable.txt* and updated the filename, allocation flags, size, and timestamps upon file creation. A subsequent filename and timestamp update were then observed once the new filename, *LO-PHI.txt*, was entered.

3.7.2 Filtering Background Noise

While the ability to provide a complete log of modifications to the entire system is useful in its own right, it is likely more relevant to extract only those events that are attributed to the binary in question. To filter out the activity not attributed to a sample's execution, we first build a controlled baseline for both our physical and virtual SUTs by creating a dataset (10 independent executions in our case) using a benign binary (*rundll32.exe* with no arguments). We then use our analysis framework to extract all of the system events for those trials and created a filter based on the events that frequently occurred in this benign dataset. Two of our memory analysis modules, *i.e., filescan* and *timers*, had particularly high false positives and proved less useful for our automated analysis. To reduce false positives in our disk analysis, we decouple the filenames from their respective master file table (MFT) record number.

3.7.3 Experiment 1: High-fidelity Output

To verify that LO-PHI is, in fact, capable of extracting behaviors of malware, we first evaluated our system with known malware samples, for which we have ground truth. In our first case study, we evaluated a rootkit that we developed utilizing techniques from *The Rootkit Arsenal* [112] (*Section* 3.7.3). Similarly, we were able to obtain a set of 213 malware samples that were constructed in a cleanroom environment, and were accompanied by their source code with detailed annotations. All the binaries in this experiment were executed on both physical and virtual machines that were running Windows XP (32bit, Service Pack 3) as their operating system.

Homemade Rootkit

Our rootkit stealths itself by adding hooks to the Windows Global Descriptor Table (GDT) and System Service Dispatch Table (SSDT) that will hide any directory or running executable with the prefix *$$_rk* and then opens a malicious FTP server. The rootkit module is embedded inside a malicious PDF file that drops and loads a malicious driver file (*hookssdt.sys*) and the FTP server executable (*$$_rk_sketchy_server.exe*). *Figure* 3.6 shows the complete post-filtered results obtained when running this rootkit through our framework. Note that we received identical results for both virtual and physical machines, which exactly matches what we would expect given our ground truth. We clearly see our rootkit drop the files to disk (*Figure* 3.6*f*), load the kernel model (*Figure* 3.6*d*), hook the kernel (*Figure* 3.6*e* and *Figure* 3.6*c*), and then execute our FTP server (*Figure* 3.6*a* and *Figure* 3.6*b*). We have omitted the creation of numerous temporary files by Adobe Acrobat Reader and Windows as well as accesses to existing files (81 total events) in *Figure* 3.6*f* to save space, however all disk activity was successfully reconstructed. Note that we can trivially detect the presence of the new process as we are examining physical memory and are not foiled by execution-level hooks.

We also ran our rootkit on the Anubis and Cuckoo analysis frameworks. Anubis failed to execute the binary, likely due to the lack of Acrobat Reader or some other dependencies. Cuckoo produced an analysis with very similar file-system-level output to ours, reporting 156 file events, compared to our 81 post filtered. However, we were unable to find our listening socket, or our GDT and SSDT hooks from analyzing their output. While our FTP server was definitely executed, and thus created a listening socket on port 21, it is possible that our kernel module may not have executed properly on their analysis framework. Nevertheless, we feel that our ability to introspect memory to find these obvious tells of malware, is a notable distinction. Subsequently, the lack of execution for such a simple rootkit also emphasizes the importance of having a realistic software environment as well as a hardware environment. We attempt to address this issue for our analysis in *Section* 3.7.5.

Labeled Malware

For the analysis of our 213 well-annotated malware samples, we first performed a blind analysis, and then later verified our findings with the labels. Note that there were samples that exhibited more behaviors than those listed here, only the most interesting findings are shown.

VM-detection We found that 66 of these samples were labeled as employing either anti-VM or anti-debugging capabilities. However, none of the 66 anti-VM samples performed QEMU-KVM detection; instead they focused on VMWare, VirtualPC, and other virtualization suites. As expected, all of the samples executed their full payload in both our virtual and physical analysis environments.

New Processes We found that 79 of the samples created new long-running processes detected by our memory analysis. The most commonly created process was named *svchost.exe*, which occurred in 15 samples. In addition, 2 other samples had variations of *svchost.exe*, *i.e., dddsvchost.exe* and *cbasvchost.exe*. These 17 samples dropped their own *svchost.exe* binary to disk, which was detected by our filesystem analysis, and executed the binary, which opened up a TCP listening socket on port 1053. Port 1053 is associated with the "Remote Assistance" service by the Internet Assigned Numbers Authority (IANA). The second most common process was named *bot.exe* and occurred in 12 samples, and 4 of these 12 samples also had the third most common process, which was named *dwwin.exe*. The *dwwin.exe* binary claimed to be Dr. Watson, a debugger included in Windows, but also appeared to be injected with malicious code. The 4 samples each created 2 UDP listening sockets on ports 1045 and 1046, one owned by *bot.exe* and the other owned by *dwwin.exe* respectively. We inferred from this behavior that these two groups of samples were derived from the same two malware families and contained remote administration tools (RATs), which we confirmed with the ground truth labels.

We also found 3 samples that executed the SUT's legitimate *firefox.exe* browser, but loaded with a suspicious library *needful.dll* that they dropped to disk. The *firefox.exe* process opened TCP listening sockets on ports 1044 and 1045 in 2 of the 3 samples, suggesting that these samples were also RATs attempting to masquerade as the Firefox browser. This supposition was also confirmed by the ground truth data.

Data Exfiltration We successfully detected 46 samples that attempted to collect and exfiltrate data through a combination of our disk and memory analysis. We initially flagged 2 particular samples because they appeared to be exfiltrating data over external IPs over port 25, which is reserved for the Simple Mail Transfer Protocol (SMTP). Our disk analysis of these samples showed a number of suspicious file reads, including reads of Firefox's *cert8.db* and *key3.db* for all user profiles stored on the SUT. These files store user installed certificates and saved passwords respectively, and there were no Firefox processes running during the execution of those samples. Searching for similar suspicious disk behavior in the rest of the labeled set yielded 44 additional samples that appeared to be exfiltrating data. Again, our detections correctly matched the ground truth data.

Worms and Network Scanning We detected approximately 30 labeled samples having worm propagation and network scanning behavior, which was also confirmed by the ground truth data. These samples contacted a significant number of IP addresses and opened up a large number of network sockets in our five minute window. For example, 8 of the samples contacted over 140 IP addresses, and 13 samples opened more than 2000 sockets. The same 13 samples appeared to target external IPs over port 135, which is associated with Microsoft RPC, a service that has had remote exploitable vulnerabilities targeted by worms in the past.

Command and Control (C2) and DNS We detected 14 samples that attempted to contact external servers over TCP port 6667, which is associated with the Internet Relay Chat (IRC) protocol. IRC is also commonly used as a C2 mechanism for remotely controlling malware, which was the case for these samples as confirmed by the ground truth data. The most common DNS queries were for the hostnames *579.info* (55 samples), *windowsupdate.net* (16 samples), *time.windows.com* (11 samples), *wpad* (11 samples), and *google.com* (10 samples). The ground truth data indicated that some of these queries were intended as red herrings while other queries were for actual contact with more suspicious hostnames such as *irc.site406.com*, *asdf.it*, etc.

Kernel Modules We detected 3 samples that unloaded the *ipnat.sys* driver and appeared to gain persistence by replacing it with a malicious version.

3.7.4 Experiment 2: Unlabeled Malware

In this experiment, we demonstrate our framework's ability to scale and extract useful results from completely unknown malicious binaries, which were obtained from the same source as the labeled data and also said to target Windows XP. The physical SUT was the same as described previously (Dell T7500 with 1GB of RAM) but the virtual machines were instantiated on a server with six quad-core Xeon X5670s (24 logical cores) and 68GB of RAM. This enabled us to instantiate a pool of 20 virtual machines with instrumentation. Due the vast difference in runtimes and resources, we were able to run far fewer samples in our physical environment. We ran 1091 samples in both environments before running out of available storage for our data on our development server. We present the general types of behaviors detected by LO-PHI in this section. Without ground truth data or manual reverse engineering, we are unable to verify any strong claims from our findings—however, we feel that the findings clearly demonstrate the usefulness of our system. Basic statistics for our analysis of these unlabled samples are shown in *Table* 3.1.

New Processes A large majority (70%) of the wild samples created new processes that persisted until the end of our analysis. The most common names are shown in *Table* 3.2. Unsurprisingly, most of the malware appeared to either start legitimate processes or

Table 3.1: Overall statistics for unlabeled malware (*Section* 3.7.4).

Observed Behavior	**Number of Samples**
Created new process(es)	765
Opened socket(s)	210
Started service(s)	300
Loaded kernel modules	20
Modified GDT	58
Modified IDT	10

masquerade as innocuously named processes. We discovered 4 samples that started a process with the same name as the currently logged in user. We found 11 samples created at least 10 new processes on the SUT, one of which created an unusual 84 new processes.

Table 3.2: Top processes created by wild malware (*Section* 3.7.4).

New Process	**Number of Samples**
IEXPLORE.exe	31
dwwin.exe	30
svchost.exe	30
explorer.exe	14
urdvxc.exe	13
dfrgntfs.exe	13
wordpad.exe	12
defrag.exe	12

Sockets About 19% of the wild samples opened at least one network socket. The most commonly opened sockets are shown in *Table* 3.3. Three samples stood out as potential worms or network scanners as they created over 1900 sockets; the next highest sample created a mere 44 sockets. Unlike our labeled set, none of the wild malware seemed to use obvious C2 channel ports such as 6667 (IRC). For example, only one sample sent traffic over port 80.

Table 3.3: Top 6 sockets (by port and protocol) created by wild malware (*Section* 3.7.4).

Port	**Protocol**	**Number of Samples**
1038	UDP	58
1039	TCP	42
1042	TCP	37
1038	TCP	36
1040	TCP	36
1041	TCP	32

Services About 27.5% of the wild samples started and installed at least one new system service. Most of these services suspiciously claimed to be hardware drivers such as USB or audio drivers. For example, over 250 samples loaded a driver claiming to be *hidusb.sys* (for Human Interface Devices over USB), possibly as an attempt to perform key logging.

3.7.5 Experiment 3: Evasive Malware

In this section, we exhibit LO-PHI's ability to analyze evasive malware, which thwart existing analysis frameworks. Because we aim to analyze modern malware samples, we ran these analyses on the same hardware, but with Windows 7 (64-bit) as our operating system. Subsequently, we also installed numerous potentially vulnerable and frequently targeted applications [113]. Specifically, Acrobat 9.4.0[1], Flash 10.1.85.3, Java 7u0 (64-bit), Firefox 38.0.1, Chrome 43.0.2357.64 (64-bit), .NET 4.5.2, and Python 2.7 (64-bit). The analysis was done exactly as described above. However, the Volatility modules used were limited to those that supported Windows 7, from which we selected the following to use in our analysis: psscan, envars, ssdt, netscan, ldrmodules, driverirp, and psxview (See *Figure* 3.4). It is worth noting that the ssdt and driverip modules did not return any findings in our dataset.

[1]This was the last release before strict sandboxing.

Paranoid Fish

First, we highlight our ability to analyze evasive binaries with a ground truth sample. We chose Paranoid Fish (*pafish v054*) [114], a proof-of-concept open-source tool that demonstrates various VM detection and anti-debugging techniques used by actual malware. When pafish is executed, it writes a file to disk for each artifact that it observes. Pafish is currently able to detect most popular analysis frameworks. For example, when run against Anubis [5], pafish dropped the following files:

hi_qemu
hi_sandbox_NumberOfProcessors_less_2_GetSystemInfo
hi_sandbox_NumberOfProcessors_less_2_raw
hi_sandbox_drive_size
hi_sandbox_drive_size2
hi_sandbox_mouse_act
hi_sandbox_physicalmemory_less_1Gb

Similarly, Cuckoo Sandbox had the following artifacts:

hi_CPU_VM_rdtsc
hi_CPU_VM_rdtsc_force_vm_exit
hi_sandbox_mouse_act
hi_sandbox_drive_size
hi_sandbox_drive_size2
hi_hooks_deletefile_m1
hi_virtualbox

On the contrary, when executed in our physical analysis environment, the only artifacts that pafish detected were:

hi_sandbox_physicalmemory_less_1Gb
hi_sandbox_drive_size2

In this instance, we were using a 750 GB hard drive with a 10 GB partition on it and 1 GB of physical memory. These artifacts are very easily removed by simply adding more hardware to the SUT. However, this will have a direct effect on the time per sample in an automated environment, as a larger disk image will have to be restored, and larger memory snapshots will require more time and space. Nevertheless, LO-PHI is able to analyzing pafish without detection.

Labeled Malware

To evaluate LO-PHI's ability to analyze real malware samples, we obtained a set of coarsely-labeled evasive malware samples, generated by Kirat *et al.* [16] in their previous work. Because these samples were specifically labeled as evasive, we only present the findings from executing them in our physical environment. While we had ground truth that these samples employed evasion techniques, capable of evading most popular analysis frameworks, we were not given the intended effect or target operating system of the samples, as we were with the samples in *Section* 3.7.3. Similarly, because of our aforementioned networking restriction, we expect that numerous samples will produce uninteresting behavior without access their command-and-control infrastructure. Thus, we are unable to make any definitive claims as to specific intent of the malware. We present our aggregated findings below, which indicate that our framework successfully avoided their evasive behaviors. The dataset consisted of malware labeled as using the evasion techniques outlined in *Table* 3.5.
A summary of our findings is presented in *Table* 3.6.

Wait for keyboard Due to the small number of samples employing this type of technique, we were not able to draw any interesting conclusions from these samples, however all of them appeared to execute successfully. One presented an error dialog window that our framework was able to locate and click, which appeared to kill the sample. This particular sample also made a DNS query to `goldcentre.ru`. The other two had no notable effects on our system.

BIOS-based All of the examples in this category appeared to trigger their payload. That is, they were unsuccessful in detecting our analysis framework, and exhibited some interesting behaviors. Every sample attempted to create an output network connections to `smtp.mail.ru`. Two of them attempted to determine their IP addresses using “whatismyip” services. The samples also spawned new processes that persisted throughout our analysis, most masquerading as existing Windows services. The `psxscan` module indicated that the processes `122.exe` and `123.exe` were spawned in two cases, `explorer.exe` was also spawned by two of the samples. Most interestingly, one of the samples created a hidden `svchost.exe` which was invisible to every process enumeration method except `psscan`.

Hardware-id-based These samples also exhibited interesting behaviors. Most notably, 23 of them started `TrustedInstaller.exe`, while 25 of the original processes continued running for the duration of our analysis, and the others appeared to spawn new processes. All of the samples also attempted to reach out to network resources: 24 of them attempted to connect to `219.235.1.127:80`, 1 attempted to connect to `62.75.235.238:443`, and 2 attempted TCP connections to either `8.8.8.8` or `8.8.4.4`, both Google-owned DNS servers, on port 53, which is the DNS port for UDP communications. All of the samples imported at least 32 modules, with the most active sample importing 156 unique modules. Finally, 11 of them appeared to present buttons that were detected and clicked by LO-PHI, and 2 of them set particularly interesting environment variables `9Yy9Y9YYy9YYy` and `YYY9YYY9YYY99`, which both had the value of `E4EC4E2160D8E128C919C56915BFED6C`.

Processor feature-based These samples produced the least compelling findings. While most of them persisted, or installed new processes, 11 had no new processes in memory. Those that did spawn new processes had filenames similar to before, with 4 of them once again loading `TrustedInstaller.exe`, 3 starting a more stealthy `netsh.exe`, 1 spawning a malicious `taskhost.exe`, and, perhaps the least stealthy sample, launching `trojan.exe`. Most of them also exhibited network activity, primarily DNS traffic, with 8 of the samples querying a variation of `boxonline`, and 7 of the samples attempting reach port 8 on various IP addresses. More interestingly, one of the samples attempted to contact 219.235.1.127, and then opened a local listening socket. A single sample in this set also set the `SEE_MASK_NOZONECHECKS` environment variable to “1”, which is a variable that will hide security warnings in Windows XP. This leads us to believe that at least some of the malware in this set was targeting an older version of windows, and likely explains why some of the samples appeared to have no effect. Two of samples also presented dialog boxes and the button “OK” was clicked.

Exception-based The exception-based malware samples also exhibited similar behavior, with all but 3 of the samples spawning new processes or continuing to execute for the duration of our analysis. Unsurprisingly, many of these samples also attempted to engage the network. There appeared to be two distinct clusters that reached out to various domains with the strings `boxonline` (31 samples) and `backupdate` (26 samples), with the others calling out to unique domains. The “boxonline” samples indicate that these may be the same class of malware that was previously observed in the processor-feature-based samples. Again, a few of the samples appeared to present a graphical interface with the text “OK,” which was successfully clicked.

Timing-based This was our largest dataset, and thus yielded the most diverse findings. Again, a majority of the samples (193 out of 251) spawned new processes or persisted throughout our analysis. The most interesting process names being: `skype.exe`, which was launched by one process and also hidden from normal windows process enumeration; `taskhost.exe`, which was spawned in a hidden state by 22 processes and a less-stealthy manner by 10 other samples; `conhost.exe`, which was also spawned in a stealthed state; and one sample spawned `facebook.exe`. Once more, we saw 4 samples set the `SEE_MASK_NOZONECHECKS` environment variable, indicating that Windows XP was likely their intended target. This dataset also had a significant number of samples (156) making `boxonline` DNS queries, and 5 of the samples querying `backupdate`. None of these samples produced network traffic aside from DNS.

While our analysis did not indicate malicious behavior in all of the samples in this dataset, we were able to detect typical malware behavior from a large majority. Some of findings indicate that at least some of the samples were targeting Windows XP, which could explain the lack of anomalies for the few that appeared benign. Nevertheless, we feel that our findings are more that sufficient to showcase LO-PHI's ability to analyze evasive malware, without being subverted, and subsequently produce high-fidelity results for further analysis. In fact, behaviors like unlinked `EPROCESS` entries and listening sockets can be exceptionally difficult to detect with software-based methodologies. Because LO-PHI has a complete view of the entire memory space and disk activity, the ability for the malware to hide its presence is greatly hindered.

3.8 Future Work

We have identified numerous areas that we feel are critical to eventually achieve a more transparent and robust framework. As previously discussed, our current approach has numerous limitations with smearing and incomplete views of the system state. CPU debuggers could alleviate these pains by either completely halting the system during memory acquisition or simply providing more insight into the internal register values. We have been experimenting with Intel's eXtended Debug Port (XDP) and ARM's DSTREAM debugger and found them to extremely powerful devices. Utilizing these hardware debugging technologies in the context of malware analysis could provide very high fidelity data while maintaining the transparency that we require. We plan to explore these techniques further and incorporate any useful developments into our framework.

In this work, we limited our scope to the system-level analysis that was provided by Volatility (*e.g.,* process list, services, sockets). While these modules are more than sufficient for the experiments proposed in this paper, we feel that expanding this scope and introspecting into an individual process's memory space to monitor process-level data (*e.g.,* stack, heap, call trace) could prove invaluable when analyzing advanced malware.

Since LO-PHI currently instruments the lowest-level of instrumentation that we are aware of, we feel that continuing to push this boundary is going to be critical for the analysis of more sophisticated future malware. To this end, we feel that our disk, memory, and CPU introspection capabilities positions us well to begin investigating malware that attempts to infect the BIOS or peripherals on a SUT for persistence and plan on continuing to develop these capabilities.

3.9 Conclusion

We presented LO-PHI, a novel framework capable of instrumenting physical and virtual machines without any software on the system, using a set of sensors and actuators. Furthermore, we developed a supporting framework capable of automating dynamic analysis of arbitrary binaries by introspecting into the memory, disk, and network activity, reconstructing the semantic operations that occurred, and outputting them as concise events (*e.g.,* process appeared, file written). We show that the sensors used to collect the necessary data produce minimal artifacts to any software running on the machine (*Section* 3.4) and that our lack of artifacts enables LO-PHI to analyze particularly sophisticated malware samples with relative ease. As malware continues to advance and evade detection, we expect hardware-based analysis frameworks to become increasingly important. We believe this work exhibits the usefulness of physical-machine introspection and instrumentation, as well as the value of forensic-based malware analysis. We demonstrated that LO-PHI provides valuable analytical capabilities that are unavailable using existing tools. To this end, we hope to engage the community by open-sourcing the project to help advance the state of the art in malware analysis.

Table 3.4: Description of Volatility modules used for evaluating evasive malware.

psscan	Enumerates processes using pool tag scanning. (Capable of finding processes that have previously terminated (inactive) and processes that have been hidden or unlinked)
envars	Extracts environment variables from processes in memory.
ssdt	Lists the functions in the Native and GUI SSDTs.
netscan	Enumerates network sockets using pool tag scanning.
ldrmodules	Enumerates modules in the Virtual Address Descriptor (VAD) and cross-references them with three unique PEB lists: InLoad, InInit, and InMem.
driverirp	Enumerates all DRIVER_OBJECT structures in memory
psxview	Helps detect hidden processes by enumerating PsActiveProcessHead using the following methods: PsActiveProcessHead linked list, EPROCESS pool scanning, ETHREAD pool scanning, PspCidTable, Csrss.exe handle table, and Csrss.exe internal linked list.

Table 3.5: Evasive malware dataset.

Technique Employed	**# Samples**
Wait for keyboard	3
Bios-based	6
Hardware id-based	28
Processor feature-based	62
Exception-based	79
Timing-based	251

Table 3.6: Summary of anomalies detected in Volatility modules and GUI buttons found in our evasive dataset when executed in our physical environment on Windows 7 (64-bit).

		Volatility Module				
		envars	**netscan**	**ldrmodules**	**psxview**	**buttons**
Malware Label	Keyboard	0	3	1	0	1
	Bios	3	6	6	6	0
	Hardware	28	27	28	26	11
	Processor	53	54	59	51	7
	Exception	76	79	77	76	7
	Timing	229	247	231	239	4

Chapter 4

Enabling Full-system Emulation of Embedded Systems

4.1 Introduction

When presented with a system to analyze, the first step usually involves performing dynamic analyses to understand how the system works. Unfortunately in the world of embedded systems, which are the most prolific type of computing devices today, this simple step is not trivial, and, in many cases, it is prohibitively expensive. In fact, no approach exists today that can emulate embedded systems in the general case. This problem stems from a few key differentiators in the embedded world: *1)* embedded systems run on a variety of architectures (*e.g.,* ARM, MIPS, or AVR), *2)* some systems implement custom ISAs with undocumented instructions, *3)* embedded software has strict hardware dependencies with little-to-no hardware abstraction software, *4)* the code is typically interrupt driven, and *5)* embedded code depends on a multitude of peripherals, which are unique for each system.

While there have been multiple approaches that address various aspects of the emulation problem, it is still far from "solved." For example, researchers have put significant manual effort into systems like the Quick EMUlator (QEMU) [101] to support more architectures and instructions. Similarly, hardware-in-the-loop record and replay systems (*e.g.,* Avatar [20] and Pretender [115]) are capable of replaying hardware interactions that were obtained through a debug interface. Finally, recent proposals that depend on a HAL being present (*e.g.,* HALucinator [116] and Firmadyne [23]) are able to satisfy hardware interactions by manually implementing specific routines (*e.g.,* `read_from_uart`). However, no system has addressed the problem of efficiently emulating peripherals, on which the majority of these embedded systems (and their emulation) depend. In theory, these peripherals could be manually implemented for a specific system, but the sheer number of peripherals and the rate at which new peripherals are hitting the market make manual implementation intractable.

Peripherals interactions are typically implemented directly as memory-mapped input and output (MMIO), where interactions with the peripheral appear as normal memory reads and writes. While traditional operating systems (OSes) implement a HAL to interact with hardware, in the form of drivers and common interfaces, embedded systems typically have their own custom HAL or interact with the peripherals directly. Herein lies the problem — the values returned from an MMIO read are unknown to any analysis that does not have access to the hardware. As such, the analysis must over-approximate the range of possible values, or infer them through best-guess heuristics by analyzing the firmware mounted by the considered system. For static analysis, this over approximation might introduce significant overheads, which can make the analysis intractable [117], and cause a loss of precision (*e.g.,* due to numerous spurious program states that would not be possible during any real execution of the system). In the case of dynamic analysis, the inability to return an expected value will likely result in the execution stalling indefinitely or the firmware entering some error-handling code. Either way, the analysis would likely fail to delve deep into the firmware code.

To make matters worse, most embedded systems are heavily dependent on interrupts that are issued from external peripherals (*e.g.,* when data has arrived) to advance their execution. In fact, it is common for embedded firmware to be completely interrupt-driven (*i.e.,* the firmware will *busy wait* until an interrupt is issued). An emulator that is incapable of issuing these interrupts will be unable to achieve any realistic functionality, as it will never execute the interrupt handlers.

The goal of this work is to create a software-based version of any hardware peripheral, automatically, that is capable of *con*ing the firmware into believing that the actual hard*ware* peripheral is present by returning valid MMIO values and issuing relevant interrupts.

Conware works by first ingesting logs of real hardware interactions (*i.e.,* interrupts, reads, and writes) of the peripheral in question, which can be obtained by using our novel source-code instrumentation or existing hardware-in-the-loop techniques (*e.g.,* Avatar [118]). These recordings are converted into directed acyclic graphs (DAGs), where the edges are annotated with MMIO writes and the interrupts and memory values are encoded as a peripheral *state* in the nodes. Then, the DAGs (one per peripheral) are converted into ω-automata using a novel graph-transformation technique, which serves as a generalized representation of the peripheral. Additionally, multiple recordings can be *merged* to create an automaton that accurately represents and generalizes *all* of the recorded interactions.

By representing peripherals as composable automata, Conware is able to not only combine recordings of the same peripheral, but it can also be used to merge disjoint peripherals to create a complete system that has multiple independent peripherals attached. For example, an IoT camera may consist of an ARM processor, a camera, a microphone, and a WiFi controller. To emulate this camera's firmware, one could purchase those same components and connect them to the appropriate ARM processor on a development board. Models for each of these peripherals could then be generated, independently, by running the example source code that is provided with the different peripheral components. These recordings could then be converted into automata, combined, and attached to an emulation environment that can be used to successfully emulate the firmware of the camera, which was never instrumented. To the best of our knowledge, CONWARE is the only system capable of creating generalized peripheral models that can be used on multiple firmware samples.

In summary, we claim the following contributions:

- an LLVM-based tool to automatically instrument source code to record all peripheral interactions (*i.e.,* MMIO and interrupts) on embedded systems,
- a novel technique for generating ω-automaton models of peripherals,
- a novel technique for *merging* peripheral models to facilitate portable models and full-system emulation,
- CONWARE: an open-source framework[1] for recording, modeling, and emulating embedded peripherals, and
- an analysis of CONWARE on popular embedded peripherals, demonstrating its efficacy by successfully modeling 10 peripherals and emulating one peripheral-heavy firmware that was never instrumented.

4.2 Background

Generally, the goal of any analysis tool is to observe the system under analysis (SUA) in a realistic environment to extract features and draw conclusions about its inner work-

[1]`https://github.com/ucsb-seclab/conware`

ings. Thus, an effective embedded systems analysis framework must create a *realistic* representation of the hardware to ensure that the software will execute correctly and produce useful results. In the literature, this concept is known as *survivability*, which is "the ability for the firmware to execute the same regions of code as it would if the original hardware were present, without faulting, stalling, or otherwise impeding this process [115]" or "the firmware never ... *crashes, stalls, or skips operations* due to peripheral IO errors. [119]".

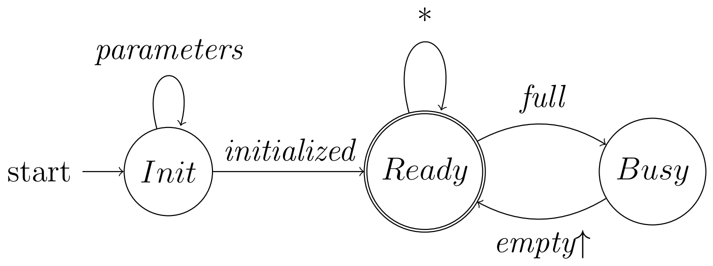
There are four general approaches to achieve survivability:

Hardware Debugging. Hardware debugging is capable of running the SUA on the actual hardware and debugging the real system. This can be done by leveraging existing debugging interfaces, *e.g.,* JTAG, or potentially interposing or snooping on the hardware components themselves. On production devices, it is increasingly rare for these interfaces to be exposed. Moreover, in many cases it is prohibitively expensive to acquire the SUA itself, instrument it, and potentially replace it if the analysis go awry — imagine irreversibly damaging an electric car.

Hardware-in-the-Loop Emulation. Debugging directly on the hardware can be slow, and lack certain features to aid analysis (*e.g.,* the ability to record every instruction or set a large number of breakpoints). Thus, hybrid, hardware-in-the-loop techniques [17, 20] have emerged to address this by emulating the main central processing unit (CPU) and intelligently forwarding any interactions with hardware peripherals to the actual hardware. While this approach offers a more feature-rich analysis, and potentially much faster, analysis environment, hardware-in-the-loop does not scale, as the number of analyses executing is still limited by the physical hardware devices that are available. This lack of scale is due to the fact that a *real*, production hardware system is needed for each emulation platform, versus emulation which scales as a factor of computation resources. Similarly, there are numerous hardware constraints that are much more difficult with hardware-in-the-loop approaches (*e.g.,* frequent peripheral-initiated interrupts [120]).

Record and Replay. This approach involves removing the hardware dependency by obtaining a high-fidelity recording of the interactions, using the actual hardware, and then use this recording to effectively "replay" the interactions with the hardware. This technique is especially useful when the portion of code being analyzed does not have many hardware dependencies or the interaction in question is being replaced by the analysis itself (*e.g.,* fuzzing [28]). This technique has been implemented by systems like PANDA [22], which record the interactions in real time and then later use this recording to feed "real" data into a more heavyweight analysis. However, record and replay techniques do not facilitate any analyses that aim to exercise hardware interactions that were not directly observed in the recording phase.

Full-System Emulation. Full-system emulation aims to provide the ability to execute the firmware in a completely emulated environment, ensuring survivability without the need for any hardware. Ideally, a full-system emulator would also enable the firmware to exercise *all* of its functionality and interact with *all* of the assumed hardware peripherals. This can either be achieved by implementing the hardware peripherals manually (*e.g.,* P^2IM [119], HALucinator [116], or Simics [121]) or automatically (*e.g.,* Pretender [115]). Despite the various advances, scalable full-system emulation of embedded systems is still an open problem. Pretender is the closest that the community has achieved, but it is only capable of replaying the basic interactions of the same hardware and firmware that were recorded in a mostly linear fashion.



$$\begin{tikzcd}
\text{start} \arrow[r] & \mathit{Init} \arrow[loop above, "\mathit{parameters}"] \arrow[rr, "\mathit{initialized}"] & & \mathit{Ready} \arrow[loop above, "*"] \arrow[r, bend left, "\mathit{full}"] & \mathit{Busy} \arrow[l, bend left, "\mathit{empty}\uparrow"]
\end{tikzcd}$$

Figure 4.1: A state-machine representation of a simple universal asynchronous receiver-transmitter (UART) controller, where the peripheral is either awaiting to be initialized, ready to received any (∗) data, or in a busy state, which is transitioned to when the buffer is full and transitioned out of when the buffer has space (potentially triggering an interrupt)

Conware has advanced the state-of-the-art in full-system emulation by facilitating *portable* and *composable* peripheral models, which can be leveraged by any emulation framework to enable unbounded execution of arbitrary firmware, even if the specific firmware and hardware being emulated have never been instrumented.

4.2.1 Motivation

As a motivating example, we examine the humble UART controller, which provides a simple interface to either read or write a single byte at a time (*e.g.,* a text-based interface).

UART, as is the case with many peripherals, has the ability to operate with or without interrupts. If an analyst uses a system that does not use interrupts to generate a model of the peripheral, it is unlikely that this model would be useful for emulating a different system that *does* use interrupts. Indeed, the interrupt-based firmware would fail to execute, as it would wait indefinitely for an interrupt to be fired. Similarly, a naïve replay of MMIO values that were recorded with interrupts would not work on a non-interrupt firmware because the observed writes (*i.e.,* enabling interrupts for cached values) would differ from the observed values (*i.e.,* busy-waiting for a *ready* bit), requiring the emulation framework to “guess” what to do next. Thus, to ensure the complete functionality of this peripheral, a valid model must account for *at least* two different scenarios, which are unlikely to ever occur in a single firmware. Even worse, UART is capable of reading and writing *any* byte. If the recorded UART interaction was non-variable (*e.g.,* it always output a fixed string) and the firmware being analyzed had more varied interactions (*i.e.,* exercising more of the peripheral's functionality), simple record-and-reply-based systems would fail to adequately handle these interactions.

Table 4.1: An recording obtained from instrumenting a simple firmware that prints ”ON\r\n” and ”off\r\n” repeatedly over UART, without interrupts

Operation	**Address**	**Value**
⋮	⋮	⋮
WRITE	0x400E0800 (control)	0x50
READ	0x400E0814 (status)	0x40001A1A (ready)
WRITE	0x400E081C (TX)	0x4F — **O**
READ	0x400E0814 (status)	0x40001818 (busy)
READ	*(repeats 434×)*	*(repeats 434×)*
READ	0x400E0814 (status)	0x4000181A (ready)
WRITE	0x400E081C (TX)	0x4E — **N**
READ	0x400E0814 (status)	0x40001818 (busy)
READ	*(repeats 2,634×)*	*(repeats 2,634×)*
READ	0x400E0814 (status)	0x4000181A (ready)
⋮	⋮	⋮

Despite the difficulties with replaying a UART recording, the actual state-machine of a typical UART peripheral is quite simple (see *Figure* 4.1). In fact, almost all embedded systems are implemented as state machines [122]. A high-level UART state machine consists of three states: an initialization state, where the firmware can set parameters like the bits of actual usable data (BAUD) rate; a ready state, where the controller is ready to receive and transmit data; and a busy state for when the controller is currently transmitting or receiving. These states are conveyed to the firmware through MMIO status registers. Thus, the firmware must either continuously check the status register before it can write new data (see *Table* 4.1) or ask the controller to trigger an interrupt when the state transitions from *busy* to *ready* (*i.e., empty*↑).

Conware is the first system capable of extracting models of the high-level state-machines that define the peripheral, using only recorded interactions.

4.3 System Design

Conware has three core components (see *Figure* 4.2):

- a source-code instrumentation framework capable of recording MMIO interactions and sending the log over any hardware interface (*Section* 4.3.1). This module is not necessary for Conware to work, as it can use recordings produced by previous work [115, 118];
- a model generation and optimization framework that converts a raw recording log into a DAG, and then into a ω-automata (*Section* 4.3.4); and
- an emulation framework that is capable of using the generated models (*Section* 4.3.7).

Indeed, these three components are standalone contributions, as each provides a unique contribution to the field.

Recording. Conware requires a recording of the low-level interactions (*i.e.,* MMIO and interrupts) with the target peripheral, which can be obtained through various methods. Indeed, hardware-based recording has already been explored, and the output of the existing tools can be used by Conware. These recordings can be of *any* firmware interacting with the peripheral; Conware does not require a recording from firmware being emulated. However, hardware-based methods require access to the original hardware *and* a debugging or instrumentation interface and are more likely to result in the

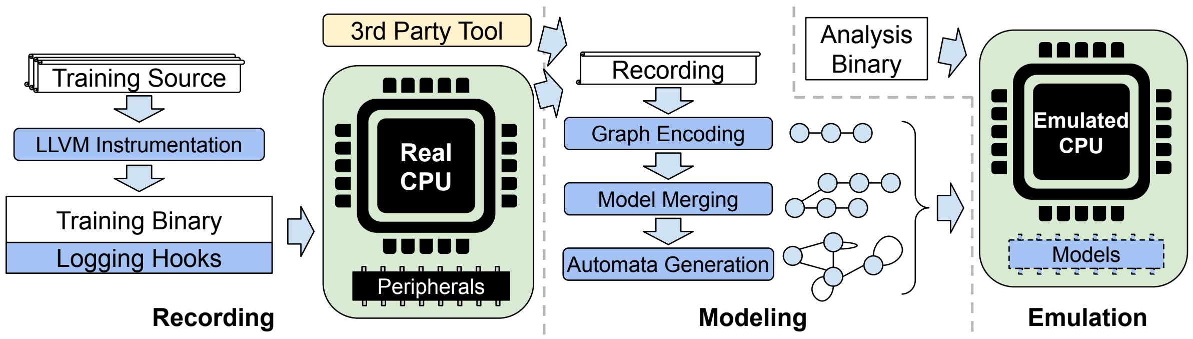


Figure 4.2: High-level design of CONWARE: logging hooks are inserted into the compiled binary, the binary is run on real hardware to extract a detailed log, these logs are then converted into concise state-machine representations, and then the uninstrumented binary and models are run on emulated hardware, enabling detailed, scalable analyses

Heisenberg effect (*i.e.*, the act of observing the phenomena alters its outcome) because of the timing overhead imposed. Thus, we created a new source-code instrumentation method for obtaining accurate hardware recordings to supplement this work.

Modeling. CONWARE generates models by first mapping the recordings of observed interactions (*raw recordings*) into directed graphs (one per peripheral), which are then converted into ω-automata. These automata are human-readable and facilitate unbounded execution since they more accurately represent the internal state machine of the real peripheral. Unfortunately, existing state machine minimization techniques (*e.g.*, the Hopcroft minimization algorithm [123], implication tables, or the Moore reduction procedure) are ill-suited for our purposes. This necessitated the creation of a novel automata-generation algorithm (*Section* 4.3.5).

Emulation. The ultimate goal is to emulate the firmware for the SUA, which can typically be obtained from the vendor's website or through more invasive techniques (*e.g.*, using a Bus Pirate [124] or advanced hardware hacking techniques [125]). Our automata, which can be generated from any example source code on any similar hardware (*e.g.*, a development board), can be plugged into any popular emulation framework as a stand in for the physical peripheral. More precisely, CONWARE is able to emulate arbitrary firmware without ever instrumenting the actual firmware or the hardware that it was intended for.

4.3.1 Source Code Instrumentation

CONWARE uses the LLVM framework [126] to instrument the firmware source code. Consequently, our instrumentation works at the LLVM Bitcode level [127] and works on all of the source languages supported by LLVM (currently over 20). The purpose of this instrumentation is to record *all* of the interactions with MMIO peripherals (*i.e.,* reads, writes, and interrupts). This is implemented as a buffered logger that exposes two functions:

- `conware_log(address, value, type)` is used to log the addresses and values that were read from or written to (*i.e.,* `type`).
- `conware_interrupt_log(num)` is used to log the firing of an interrupt of a specific interrupt request line (IRQ) number (*i.e.,* `num`).

This logging infrastructure maintains an in-memory buffer and also takes care of flushing the buffer to a known interface (*e.g.,* Joint Test Action Group (JTAG) or UART) when it is full or a programmed trigger is hit. By logging directly to memory, CONWARE incurs a minimal performance overhead, and thus a minimal Heisenberg effect. In fact, inline binary recording enables CONWARE to overcome the challenge of recording frequent interrupts accurately [115]. Furthermore, our logger provides a single place to handle multi-threading and reentrancy [128], which is necessary for accurately recording interrupts in practice. In addition to recording the immediately relevant information, we also log the program counter to facilitate debugging and future analysis.

4.3.2 Recording MMIO accesses

Though, theoretically, it is important to instrument all of the `load`s and `store`s to not to miss any MMIO access, previous work [129] has shown that MMIO is usually accessed via hardcoded addresses, which can be retrieved by analyzing the firmware code of an embedded system. Exploiting this insight provides a lower overhead and is more tractable. Thus, all of the accesses that use hardcoded addresses are instrumented by inserting a call to `conware_log` after `load`s and before `store`s, with the value being read, or that is about to be written, respectively. In fact, in our specific examples, we found that this could be optimized even further, as all of the peripheral interactions were represented as `struct`s in the source code (greatly reducing the overhead costs).

Due to memory limitations, CONWARE stores all of the reads in a compressed array format for each entry, where repeated reads are stored only once, with an associated counter. This counter and log are reset every time a new write is observed, as writes are akin to state transitions of the peripheral. This is a necessary optimization since many MMIO values are read repeatedly until they change (*e.g.,* a status register) and would quickly exhaust the buffer otherwise.

4.3.3 Recording Interrupts

To record interrupts, we first retrieve all the interrupt service routines (ISRs) along with corresponding interrupt number they service from the interrupt vector table, which is always linked at a static address. CONWARE instruments all the identified ISRs by inserting a call to the `conware_log` function at the entry of the function with the corresponding interrupt number. These interrupts are similarly compressed with a repeat counter to save buffer space and optimize our recording. The correlation between interrupts and their associated peripheral are discerned from the data sheet of the microcontroller, which are manually entered once per chipset. For example, page 38 of the datasheet for the SAM3X explicitly lists every peripheral interrupt in the nested vectored interrupt controller (NVIC) [130], and the handlers to these routines are trivially found in the source code or compiled binaries. These memory locations will be constant across all variants of the same processor (*i.e.,* all Cortex-M3 processors will have the same values [131]).

4.3.4 Encoding Recordings

Hardware peripherals typically only change states when the software *writes* a value to one of the registers on the peripheral [115] (*e.g.,* the firmware writes a *command* to the peripheral). Thus, CONWARE first encodes the recordings as simple DAGs where the edges are labeled with MMIO writes and the nodes encode the "state" of the peripheral, which includes the values that each memory region should return when they are read, as well as the interrupts that should be fired (and how many times). We call this graph a *linear model*, which is later converted into a more robust automata.

Formally, the DAG is denoted as (N, E) where E is the set of directed edges and N is the set of nodes. An edge $e_{12} \in E$ from n_1 to n_2 is represented by the tuple (n_1, n_2). Each node n has an associated state, such that $n.state \in S$, where S is the set of all states in a given DAG. This simple linear DAG can only reproduce a verbatim replay of the recorded content, as any out-of-order operations would not have a valid state transition and could only be handled by an educated guess.

As an explicit example, a write to the UART transmit (TX) register would traverse the edge with that specific value, and put the peripheral into a "new" state. This state would then return the following *pattern*: `BUSY` for the first 434 reads, and then `READY` on the 435^{th}. An explicit example of a node in our model for the UART peripheral is shown in

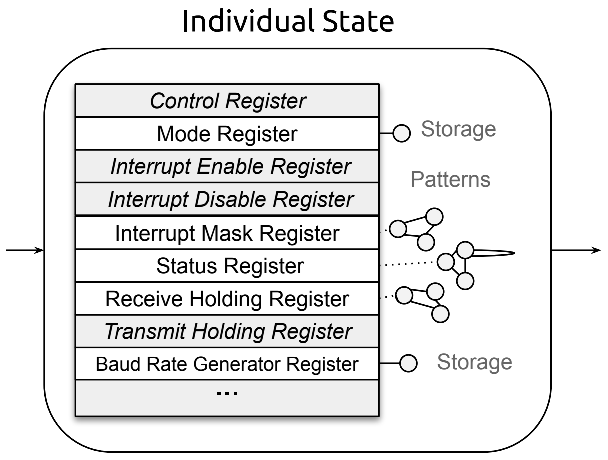


Figure 4.3: A individual node representation in our graph (each node contains a single peripheral state) of a UART controller, where edges are writes and the state encodes the values to be read from specific addresses (Storage works as normal memory and Patterns return more complex data)

Figure 4.3, where each address has its own sub-model, within the overarching peripheral graph. And an example output of converting *Table* 4.1 into our DAG representation can be seen in *Figure* 4.4.

Within in each state, each memory address is encoded as sub-model to ensure that the appropriate values are returned when that address is read. These memory models are lumped into three general types [115]:

- storage – acts like normal memory,
- pattern – a single or repeated pattern, and
- monotonic – returns a monotonically changing value

For example, if a particular address in the peripheral (*e.g.,* a status register), always returns the same value, we will simply model that as a static pattern, which will always return the same value, regardless of how many reads occur. Conversely, if the register always returns a string of `0xA`s, followed by a `0xB`, our model will keep these semantics. This is currently the state of the art [115].

Interrupts

Conware must not only support interrupts, but must be able to automatically learn *when* to trigger which interrupt. Fortunately, the NVIC is standardized for most architectures (*i.e.,* every NVIC for ARM has the same structure), which permits us to manually hardcode the appropriate actions for various timers etc. by reading the datasheet. For example, our Cortex-M3 has eight timer counters (*i.e.,* Timer Counter Channels 1 through 8), that can be enabled or disabled. Thus, when a specific timer interrupt is enabled, we can programatically trigger that interrupt periodically until the interrupt is later disabled. Somewhat unintuitively, the actually frequency of triggering the interrupt does not actually matter. For example, if the interrupt is supposed to trigger every 50 *ms* on a real board, deviating from this is unlikely to result in an erroneous emulation. The reason for this is that embedded systems rely on these interrupts for their time, and have no other timekeeping mechanisms. Thus, the code treats the interrupts as a single "time unit," but does not make any assumptions as to the actual time that has passed. This assumption is only true of timer-like interrupts.

For peripheral-triggered interrupts (*e.g.,* a *data ready* interrupt), this problem is exacerbated by the fact that interrupts can depend on the context of the peripheral and the firmware. To ensure that interrupts are triggered at the correct time, they are encoded as part of a node's state (*i.e.,* the specific IRQ numbers and how many of each), and triggered when the incoming edge is taken. Thus, interrupts will only be triggered after a write was observed that was also immediately followed by interrupts in the recordings (*i.e.,* that specific state in our model was reached, not just the address/value pair). This is contrary to previous work [115], which would observe a specific write and then begin triggering interrupts indefinitely, which is likely to over approximate in practice. When observing our UART controller, the interrupt was never explicitly disabled, but

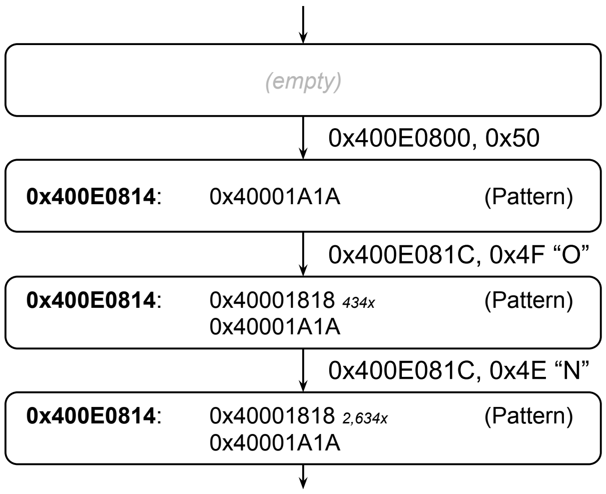

(empty)
0x400E0800, 0x50
0x400E0814: 0x40001A1A (Pattern)
0x400E081C, 0x4F "O"
0x400E0814: 0x40001818 434x 0x40001A1A (Pattern)
0x400E081C, 0x4E "N"
0x400E0814: 0x40001818 2,634x 0x40001A1A (Pattern)

Figure 4.4: A DAG representation of a simple UART device, where each node represents a state and encodes the address to be read from (*e.g.,* status register) and the values to be returned (*i.e.,* `TXRDY` or `BUSY`). Nodes are traversed when writes are observed (*i.e.,* writing "O" to the TX register)

one interrupt per write was issued (*i.e.,* each write queues a future interrupt).

An explicit example of this can be seen in *Figure* 4.5, which was generated by recording and optimizing a firmware that prints "Knock!\r\n" every time that a piezo transducer is *knocked*. Our model correctly encodes the interrupts into the state where the final byte was written, and all subsequent byte writes to the `TX` register are encoded as a self-loop. In practice this corresponds to our emulator returning busy the appropriate number of times, and triggering a *ready* interrupt every time a valid write to the TX register is observed. Indeed, this automata can be used to emulate any firmware that supports interrupts. If the execution deviates from our model (*i.e.,* the execution wrote a value that is not present in our model), we perform a breadth-first search (BFS) to identify an acceptable state within our model (*e.g.,* any write to the TX register other than "K" would ultimately take the wildcard edge where the buffer is `READY`). This, as shown in *Section* 4.4.3, will result in the proper actions ultimately being taken (*e.g.,* when enough buffered writes are observed, it will transition into the *ready* state and the interrupts will fire until the buffer is emptied).

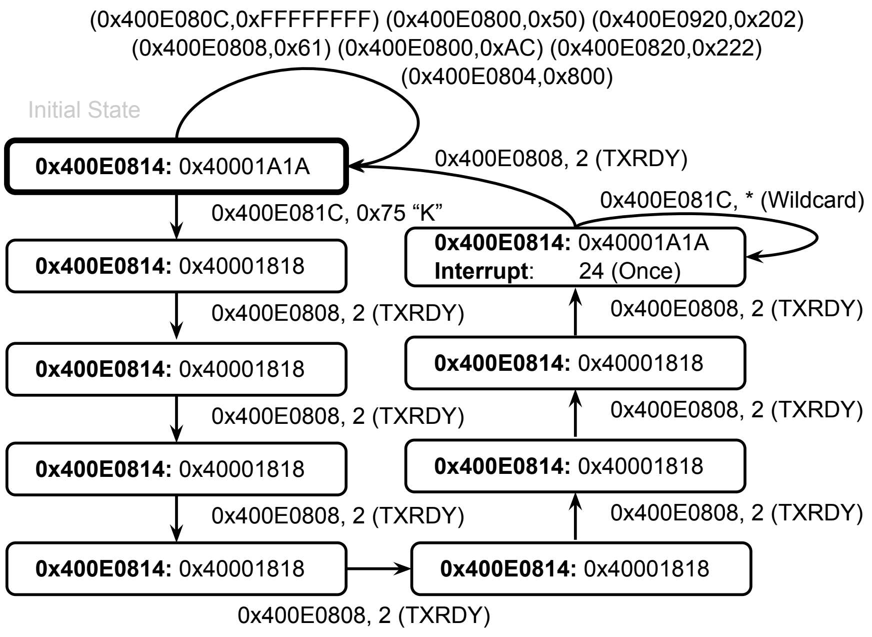


Figure 4.5: An ω-automata for a UART peripheral generated from a recording that prints "Knock!\r\n" repeatedly. The initial state will accept all configuration parameters, once a "K" is written, the peripheral becomes `BUSY` until the `READY` state is reached and interrupts are thrown.

4.3.5 Automata Generation

The next step of our approach involves transforming our DAGs, obtained as explained in *Section* 4.3.4, into ω-automata. Before generating our ω-automata we must first define what it means for two states to be *mergable* (*i.e.,* they can be merged into one), or, put another way, which nodes in the graph are in the same equivalence class. Two generic states a and b can be merged, indicated by $a \cup b$, if they are *equivalent*. The states a and b are equivalent, indicated as $a \simeq b$, if they have the same identified type (*e.g.,* storage, constant, or pattern) for every overlapping memory address, and those types are also equivalent (*i.e.,* they encode the same data). Nodes are mergable if and only if their states are mergable. Additionally, we consider two edges to be equivalent, also denoted by $\simeq$, if they have the same labels (*i.e.,* the same write address and value).

For example, take states $a, b \in S$ where a has memory models for address $0x100$

(Storage) and $0x200$ (Pattern) and b has memory models for $0x100$ (Storage) and $0x300$ (Pattern). These states would be considered to be mergable, since there is no risk of returning a wrong value. The returned values would be the same for both a or b and $a \cup b$. Indeed, $a \cup b$ is strictly more verbose than either of the individual states. Patterns are only considered equal if they are identical. While this could potentially be relaxed, there are numerous cases where the exact values are critical, *e.g.,* the NEC intermediate representation (IR) encoding protocol.

While this definition of *mergability* is intuitive, its lack of transitivity does slightly complicate the state-reduction phase. More precisely, it is possible for $A \simeq B$ and $B \simeq C$ but $A \cup B \not\simeq C$ (*e.g.,* A:[$0x100$:Storage], B:[$0x200$:Pattern], C:[$0x100$:Pattern]). Thus, we must be diligent when merging equivalence classes to make sure that all of the merges will succeed without violating the soundness of our model.

The goal of the automata-generation phase is to combine all of the mergable states to create a more general representation that can be used for indefinite execution and handle out-of-order operations. For example, a linear DAG, which is the current state of the art [115], is incapable of handling execution beyond the last node in the graph and would completely fail if firmware were to execute functionality in a different order. A generalized automata, in theory, should not suffer any of these shortcomings.

Our automata-generation algorithm starts at the initial node (*i.e.,* the node that corresponds to the *first* action in the recording) in the graph (i), and traverses the graph using a nested depth-first search (DFS) such that i is compared to every node that is reachable from i. This search is then repeated for every subsequent node in the graph, until a set of nodes that can be merged (*i.e.,* in the same equivalence class, C) is successfully identified. If two nodes are mergable, the algorithm then traverses all of the equivalent outgoing edges (*i.e.,* the edges have the same labeled memory write address and value) recursively to ensure that after the two nodes are merge that all of the edges will remain valid. More precisely, for two nodes to be merged, they must be mergable, and all of the nodes on the same outgoing edges must also be equivalent. This recursive comparison is done by first identifying equivalent edges for the two initial nodes, and then recursively identifying all of the equivalent edges for any other identified nodes until a cycle is completed or the two nodes in question share no common outgoing edges. If two unequal nodes are found, the nodes are marked as *unmergable*. Finally, because our equivalence comparison is non-transitive, we confirm the equivalence of the cross product of the various node equivalence classes before merging the equivalence class into a single node and combining the relevant edges.

After a successful merge, the algorithm is then run again, starting at the initial node (i). This process is repeated until the algorithm reaches a fixed point, which is defined by reaching the end of the nested DFS for every node in the graph without any nodes being merged. The entire algorithm is shown more formally in *Algorithm* 1. This algorithm guarantees that we have obtained *an* ω-automaton, but not necessarily the *best* or *smallest* representation, as the order of operations could impact the outcome. Nevertheless, this automaton is more than sufficient for the purpose of emulating and understanding the general structure of the peripheral's internals. The results of this algorithm on a UART recording which prints "Knock!\r\n" indefinitely is show in *Figure* 4.5.

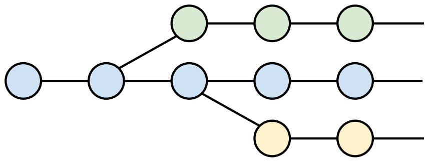
To achieve more general automata, we consider an edge to be a wildcard (*i.e.,* any value is an acceptable state transition for that address), and merge the associated edges, if and only if all of the outgoing edges with that address have the same destination node and the number of similar edges is above a threshold (*e.g.,* five). Indeed, this merging of edges greatly increases CONWARE's ability emulate unobserved code branches in our training data as well as never-before-seen firmware (see *Figure* 4.5).

```
Function GetEdges($n_1$, $n_2$):
    $s_1 \leftarrow n_1.state$;
    $s_2 \leftarrow n_2.state$;
    if $s_1 \simeq s_2$ then
        if $n_1 \in C \wedge n_2 \in C$ then
            return;
        end
        $C \leftarrow C \cup \{s_1, s_2\}$;
        $EC \leftarrow ConnectedComponents(C, n_1)$;
        $O \leftarrow OutgoingEdges(EC, n_1)$;
        $R \leftarrow \emptyset$;
        forall $e_1 \in O$ do
            forall $e_2 \in n_1.edges \cup n_2.edges$ do
                if $e_1 \simeq e_2$ then
                    $R \leftarrow R \cup (e_1.dest, e_2.dest)$;
                end
            end
        end
        return $R$;
    else
        return $\perp$;
    end
Function GenerateAutomata($N$):
    forall $n_1, n_2 \in N \mid n_1 \neq n_2$ do
        $C \leftarrow \emptyset$;
        $M \leftarrow$ GetEdges$(n_1, n_2)$;
        while $x, y \leftarrow pop(M)$ do
            $M \leftarrow M \cup$ GetEdges$(x, y)$;
        end
        if $M \neq \perp$ then
            Merge($C$);
        end
    end
    return;
```

Algorithm 1: Functions for determining if two nodes are equal and can be merged, which will ultimately update the graph by merging all "equal" nodes, and all of their annotations, into a single node

4.3.6 Combining Models

Because of the way that our automata generation is implemented, merging recordings is relatively straightforward. First, we combine the initial linear DAGs by starting at the initial nodes and merging every mergable state until there is a convergence (*i.e.,* the current node is equivalent but the outgoing edges are not equivalent). Given that peripherals are expected to power on into a known state, it is unlikely to ever have a read value differ without first seeing a deviation in the written value (*i.e.,* putting the peripheral in a different state). The resulting merged graph will then have only a few nodes with multiple outgoing edges and no cycles (*i.e.,* a tree).



Regardless of the number of recordings that are merged, the automata-generation step proceeds as it did in the single-recording scenario – iteratively merging equivalent states until no more nodes can be merged. The result of this step is a model that is generalized *and* satisfies the constraints of every input model. Said another way, this model can be used to successfully emulate *any* of the original firmware, and likely many other firmware that use the same peripherals.

4.3.7 Hardware Emulation

Our current emulation framework is built in Python as a an extension of Avatar^2 and is loosely based on Pretender. CONWARE implements a custom `AvatarPeripheral` that encompasses the entire MMIO memory region, where the reads and writes interact directly with the generated models, advancing states on writes and returning appropriate values on reads. Within this class, the memory is split into individual peripherals, which can either be identified manually (*e.g.,* by reading the data sheet) or automatically [115]. Each individual peripheral has its own disjoint automaton that is actuated in isolation. Once the emulator is running, the `write` command will result in the model advancing, either to the next state if that forward edge exists, or by performing a BFS in the case that is it does not. If the BFS fails, the first fallback is to pick the node in the entire graph that has the most observed incoming edges with that address value. If a write to this address was never observed, we simply stay in the same state and create a `Storage` model for that address.

If a state is entered that has interrupts, a thread is started for each interrupt that will trigger it the appropriate number of times. This is done to ensure that the firmware still executes seamlessly, without waiting until the interrupts are handled. Similarly, this permits Conware to trigger continuous interrupts (*e.g.,* a counter that is triggered every Xms). As previously mentioned, known interrupts are hard-coded in our emulation framework and triggered when the appropriate *enable* bit is written to a specific address, while peripheral-specific interrupts are triggered during state transitions.

4.4 Evaluation

The Arduino platform proved to be a perfect testing ground for Conware — it is open source, is compatible with a large array of peripherals, and has well-documented example code for the supported peripherals. Indeed, analysis aside, Conware provides the ability fully emulate Arduino firmware *with* arbitrary peripherals, making it the first system capable of this feat. To ensure the applicability of our evaluation to real-world systems, we opted to use the Arduino Due, which has a 32 bit ARM Cortex-M3 processor (the Atmel SMART SAM3X/A [130]). To instrument the Arduino code, we modified the build environment to instrument both the Arduino environment and the program that was being compiled on top of that environment (*i.e.,* the `.ino` file). This instrumentation is capable of automatically injecting logging into any Adruino program, including library packages, and outputting the recorded log over any standard interface (*e.g.,* UART) after the specified buffer has been exhausted (*e.g.,* 2,000 entries with compression) or a triggered event was detected (*e.g.,* a button press). This buffer can be filled, emptied, and recorded indefinitely, which enables the recording of long-running or MMIO-intensive interactions.

For our evaluation we ran a spread of unique experiments to demonstrate the practicality our modeling framework:

- Recording and replaying the same firmware with a Conware model replacing the hardware (*Section* 4.4.2)
- Recording and replaying the same firmware with a *merged* and generalized automaton — multiple recordings where merged into one model and the original firmware were run against it (*Section* 4.4.3)
- Recording and replaying an "unseen" firmware with a merged and generalized automaton — the model was generated using different recordings of individual peripherals using the example source code that was provided with those peripherals (*Section* 4.4.4)

The purpose of these experiments was to demonstrate that Conware is a viable solution for emulating hardware peripherals and that it is capable of handling real-world peripherals. All of these experiments were run in a fully-automated fashion (*i.e.,* a single script was executed to generate all of the models and execute the emulator). We do not claim that these findings indicate that Conware is *the* solution or that emulating embedded systems is *solved*, but instead advocate graph-based, automata modeling of peripherals as a viable technique for the research community to focus more effort on to continue to address this critically-important problem (*i.e.,* full-system emulation of embedded systems).

```
void loop() {
  printf("ON\n\r");
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  printf("off\n\r");
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}
```

Listing 1: Simple Arduino program that blinks an LED and prints the lumination status over UART

Before delving into the results, we want to first emphasize the complexity involved with emulating an embedded system. We first instrumented an Arduino program that simply blinks the on-board light emitting diode (LED) and prints text over UART (see *Listing* 1). With a 2,000 entry recording buffer (with compression), our instrumentation logged 88,287 MMIO accesses, which consisted of 65 unique addresses across 11 peripherals. The breakdown of these accesses were as follows: 52 unique addresses were written to (159 unique address-value combinations), 21 unique address were read from (37 unique address-value combinations), and zero interrupts (excluding `SysTick`) were observed. To represent these logs in our linear DAG it requires 11 separate graphs (one for each peripheral), which in total contain 1,014 nodes, 1,003 edges, and encode 87,284 values in their states (*i.e.,* values to return when certain memory addresses are read). The UART peripheral accounts for 367 of those nodes and the platform input and output B (PIOB) peripheral, which is used to control the LED, accounted for 506 nodes. Emulating even a simple firmware, such as this one, is no trivial task. After applying our automata-generation technique, the graphs contained a combined 26 nodes (a 22× reduction) and 45 edges, 21 of which are self loops. This reduction can be made arbitrarily high by recording for longer, as the number of equivalence classes is static.

4.4.1 Dataset and Experimental Setup

To demonstrate the breadth of devices, and interactions, we strategically choose a few indicative peripherals, many of which are used by a hobbyist smart door lock firmware that we emulate in *Section* 4.4.4:

- `IR`: an IR remote controller [132] and an IR receiver,
- `LCD`: a standard HD44780 liquid crystal display (LCD) display,
- `Knock`: a piezo transducer to detect a "knock,"
- `UART`: various UART interactions, both with and without interrupts,
- `Color`: a Cadmium-Sulfide (CdS) photo resistor used to detect the color incoming light,
- `Servo` [133]: a 180 degree servo motor,
- `LED`: both onboard and external LEDs,
- `Ethernet`: an Ethernet board capable of 100 Mbit communication,
- `Button`: an external button,and
- `RF TX`: a 433 MHz radio frequency (RF) receiver.

The IR remote is particularly interesting, as it works by starting a timer (`TC5`), which will fire interrupts indefinitely. The interrupt controller reads the value from the IR receiver (*e.g.,* high or low), and will continue to receive data, according to the NEC IR transmission protocol [134], until an entire data unit was received. At this point, the individual bits in the buffer are decoded into their respective byte values (*e.g.,* the number "1" is encoded as `0xFF30CF`), which can then be parsed and handled by the application. This means that in order to properly emulate this peripheral, the interrupts must be triggered appropriately, the individual bits must be fed in correctly, and in order, and the subsequent actions must also be supported (*e.g.,* flashing an LED or printing values over UART, which also uses interrupts).

The `LCD` code is high-bandwidth, and indicative of more complicated peripherals that display detailed information to users (*e.g.,* an alarm clock, weather app, or smart electronic). The `Knock` sensor is representative of any analog sensor (*e.g.,* temperature, acceleration, or humidity) that has a range of values, of which the firmware is typically concerned with some "threshold" value. `UART` is still one of the most popular protocols for interacting with embedded systems, and presents an interesting case because, while its actual functionality is simple, the implemented functionality is unbounded (*e.g.,* complete shell interfaces). `Color` is an analog sensor that also depends on actuating nearby LEDs (*e.g.,* red, green, and blue) to detect the reflected light. Buttons, servos, and LEDs are common interfaces for most embedded systems that need to communicate with the user efficiently or actuate some external motor. And, finally, to ensure that our peripherals were indicative of popular IoT devices, which communicate with external devices, we also included an Ethernet controller and a popular 433 MHz wireless radio. Both of these communication peripherals were running echo servers and were appropriately actuated in training.

All of these sensors came with accompanying libraries and example code, *i.e.,* `File|Examples` in the Arduino integrated development environment (IDE), that was used in our experiments to remove any biases. These examples programs were used "as is" to generate our recordings. Emulations were run on a laptop with an Intel® Core™ i7-8550U CPU @ 1.80 GHz and 16GB of memory.

Real-world Relevance In addition to the chosen peripherals being popular in IoT devices, the Arduino platform is also used by many rapid prototyping companies [135]. Thus Arduino-based peripherals and their interactions in Arduino products should be indicative of real-world applications, as the major difference from prototype to production is typically cost reduction by choosing smaller, less expensive parts and creating a custom printed circuit board (PCB) that only includes the necessary components [136]. In our smart door lock firmware, the peripheral interactions are also non-trivial. The interrupts for the IR sensor are constantly firing to accept user input. If the “knock” command is received, it then enters a loop that reads the piezo transducer for a fixed amount of time (using a hardware timer), and then will actuate the servo appropriately based on the correctness of the knock pattern. Similarly, if the “color” command is received, the firmware will enter a function that illuminates three LEDs in sequence (red, green, and blue), while reading the value of the photo resistor to determine the color of the object that is near the sensor. If the correct color is detected, the servo is actuated accordingly. These peripheral dependencies, which are typical of IoT devices necessitate a high-fidelity emulation framework — a simple replay of these peripherals would not suffice in exercising any of the interesting functionality of this firmware (*i.e.,* unlocking the door).

4.4.2 Record and Replay

First, we wanted to demonstrate that Conware is able to achieve the basic record and replay functionality that existing system have focused on. In these experiments, we took the example code for our test cases and compiled both an instrumented (*i.e.,* logging enabled) and uninstrumented version of the firmware for each. The instrumented version was then executed on the real hardware with us interacting with the peripheral (*e.g.,* pressing buttons on the IR remote or knocking the piezo transducer) until the record buffer was full and the recording was dumped over UART. The recording was then converted into a linear model, and then an automaton. Replaying the linear model is effectively equivalent to the current state of the art (*i.e.,* Pretender [115]).

For each of these direct record-and-replay cases, both the linear graphs and generalized graphs were able to replay the originally recorded firmware. However, after the logs were exhausted (*i.e.,* the emulation ran for more time than the original recording), the differences were clear. In fact, without a technique like CONWARE, there is currently no proposal (aside from guessing) for how to handle future execution. Nevertheless, CONWARE's state-machine-like models were able to successfully execute indefinitely. We used the generated models to run each of the samples for 10 minutes in our emulation framework. To enable a straightforward comparison, our emulation framework outputs logs in the same format as our recordings. Thus, we are able to compare the accesses to each peripheral, in order. Since our replays are deterministic and will return the same recording every time, there is no value in running the experiments more than once.

This comparison is done by first splitting the output of each log into its respective peripheral. For example, if 11 peripherals were observed, the log would be split into 11 separate logs where the entries for each peripheral are in sequential order. The logs for each peripheral (*i.e.,* the recorded log and the emulated log) are then compared directly using sequence matching, where duplicates are treated as a single value. More precisely, any repeats reads are effectively collated into a single entry, which ensures that the same sequence, but not necessarily the same exact observation. This ensures that we do not unnecessarily punish ourselves for things like status registers, which can return the same valuable a variable number of times without impacting the code execution, but still enforces strict order, which should only be the same if the states are advancing correctly.

The results of executing each example firmware against its own automaton is shown

Table 4.2: A comparison of the in-order MMIO access logs of both the recorded and emulated firmware

Firmware	**Conflicts**	**Additional (%)**	**Missing (%)**	**Total (Emu.)**	**Total (Rec.)**
`Knock`	0 (0.000)	0 (0.000)	0 (0.000)	34,028	5,607
`UART`	0 (0.000)	0 (0.000)	0 (0.000)	653,793	222,123
`Servo`	0 (0.000)	0 (0.000)	0 (0.000)	949	4,571
`Blink2`	0 (0.000)	0 (0.000)	0 (0.000)	15,393	2,606
`Blink`	0 (0.000)	0 (0.000)	0 (0.000)	212,594	88,286
`IR`	0 (0.000)	0 (0.000)	1 (0.002)	53,955	205,977
`LCD`	10 (0.221)	56 (1.237)	136 (3.005)	533,997	4,506
`Ethernet`	0 (0.000)	1 (0.022)	16 (0.356)	153,170	4,491
`Button`	0 (0.000)	0 (0.000)	0 (0.000)	614,354	4,603
`Color`	0 (0.000)	1 (0.039)	1 (0.039)	17,237	2,570
`RF RX`	1 (0.001)	2 (0.002)	3 (0.004)	82,478	124,807

in *Table* 4.2. All but four of the firmware replayed *exactly* as they did in the recording. Three of them (*i.e.,* `IR`, `Color`, `RF RX`, and `Ethernet` had a few missing entries due to UART buffer inconsistencies) and LCD had some executions appear out of of order due interrupts arriving in a different order. Indeed, the differences indicate, more than the identical comparisons, that our automaton is better than a simple replay. Somewhat more interesting than the order of the accesses is the total number of MMIO accesses that were observed. All but two (*i.e.,* `IR` and `Servo`) actuated *far more* MMIO accesses than were observed in the initial recording, emphasizing the power of Conware. `IR`, `RF RX` and `Servo` are very `MMIO` heavy, which accounted for the lower number of observed accesses since the MMIO accesses incur a larger overhead in emulation. The same one-to-one correlation was observed beyond 10 minutes, and the values observed over UART was also identical.

As a sanity check, we attempted to execute the firmware for `Blink`, which does not use interrupts, using the model that was generated from `Knock`, which does, to see if the UART peripheral would work correctly. Unsurprisingly, this emulation failed to print any characters after the first one, since the UART status register would continually return

BUSY, expecting the firmware the buffer them and request interrupts. This experiment demonstrates the subtlety that must be accounted for when emulating embedded systems.

4.4.3 One Model to Emulate Them All

To demonstrate the efficacy of our technique and to quantify the compression that is achieved by our automata-generation phase, we examined the resulting the graphs at each step in our process. *Table* 4.4 shows the number of states (*i.e.,* nodes), edges, and self-loops for each peripheral in the case of a linear graph (*i.e.,* the current state of the art) and our automata graphs (*i.e.,* after our automata-generation step), which are denoted with a $_G$ subscript. Looking at this table, it is clear to see that our state-reduction is highly effective, reducing the number of required states by more than 10 fold in every instance. Again, these reductions can be made arbitrarily large by inputting longer recordings. Moreover, this table again demonstrates the complexity of the "re-hosting problem" (*i.e.,* emulating embedded systems). While these examples are objectively simple, they still require the proper emulation of multiple peripherals to execute successfully, even if they do not explicitly use them.

The true value of our automata-generation is not to create models that can be used beyond the recorded execution, but to create portable models by merging the recordings from various firmware to capture the full gamut of peripheral interactions, and enabled the emulation of *any* firmware. As a first step to this, we show that ConWare can generate merged models that are at least able to emulate their original recordings. This would not be possible with simple linear models alone (*i.e.,* Pretender) — at least a tree would be required to encode the divergence point. Moreover, merging multiple recordings with a simple tree would result in very large models and would be incapable of handling a firmware that actuates a mixture of the functionality observed in multiple training

Table 4.3: Summary of executing the various firmware on a merged model that is a composition of their individual recordings. Example firmware emulated for 10 minutes, smart-lock firmware executed for 60 minutes. MMIO writes, reads and peripheral-specific interrupts are reported, as well as graph traversal statistics: long jumps (took a non-existent edge), wildcards (took a wildcard edge), BFS (performed a BFS to find the appropriate next state)

Firmware	**Writes**	**Reads**	**Interrupts**	**Long Jumps**	**Wildcards**	**BFS**
`Knock`	14,112	15,562	3,288	6	427	13
`Servo`	185,067	111,791	0	3,681	16	191
`Button`	344,876	173,157	0	0	16	3,985
`IR`	228	90,179	24	0	19	0
`Blink2`	10,718	4,305	0	16	6,947	81
`Color`	1,601	1,744	226	0	36	22
`Lock`	795,597	1,598,900	397	136	17	26,895

recordings.

Our `Knock` and `IR` examples both read sensor values and report the value over UART. However, these sensors are very different, and the UART output is completely divergent, the text "Knock" versus a hexadecimal representation of the encode button press. This made these two samples an ideal testing ground for testing the portability of our models. When the models were merged, they were both able to emulate successfully, using *the same automaton.*

With the basic functionality confirmed, we merged the models for multiple non-overlapping peripherals (*i.e.,* they all use different physical pins) in an attempt to create a full-system emulation model, capable of handling *any* of the modeled peripherals. Specifically, we created a single model using the recordings from `Color`, `IR`, `Knock`, `Blink2` (which blinks external LEDs), `Servo`, and `Button`. These peripherals were chosen because they are all used by the smart lock firmware that is our ultimate emulation target, thus we refer to this automaton as `Lock`.

Indeed, we were able to use the `Lock` model to successfully emulate *all* of the original

Table 4.4: Summary of the complexity of both the linear and generalized graphs for our five indicative firmware samples, showing edges, E, self-loops, L, and nodes, N, for each peripheral. Models were generated from recordings with a 2,000 item buffer (with compression). The columns relate to the peripheral controller that the ARM processor interfaces with (*i.e.,* the actual peripheral is behind by these standard interfaces)

	UART			**PIOA**			**PIOB**			**PIOC**			**PIOD**			**UOTGHS**			**TC1**			**EEFC0**			**ADC**			**PMC**			**WDT**			**EEFC1**			**Total**		
Name	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N
IR	304	0	305	50	0	51	25	0	26	27	0	28	22	0	23	13	0	14	8	0	9	1	0	2	10	0	11	15	0	16	1	0	2	1	0	2	477	0	489
IR$_G$	128	10	103	7	3	3	4	2	2	4	2	2	4	2	2	4	2	2	3	2	2	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	171	30	128
Knock	963	0	964	50	0	51	391	0	392	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	714	0	715	12	0	13	1	0	2	1	0	2	2,190	0	2,201
Knock$_G$	11	2	9	7	3	3	126	62	62	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	705	352	352	9	3	6	1	1	1	1	1	1	873	431	441
UART	552	0	553	50	0	51	31	0	32	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	715	0	726
UART$_G$	44	22	23	7	3	3	6	3	3	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	85	41	47
Servo	7	0	8	50	0	51	25	0	26	1,842	0	1,843	17	0	18	13	0	14	459	0	460	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	2,439	0	2,451
Servo$_G$	1	1	1	7	3	3	4	2	2	274	137	137	4	2	2	4	2	2	203	4	69	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	514	158	228
Button	7	0	8	50	0	51	2,451	0	2,452	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	2,591	0	2,602
Button$_G$	1	1	1	7	3	3	813	406	406	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	850	423	428
Blink	366	0	367	50	0	51	505	0	506	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	1,003	0	1,014
Blink$_G$	4	2	2	7	3	3	6	3	3	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	45	21	26
RF RX	97	0	98	50	0	51	25	0	26	57	0	58	52	0	53	13	0	14	-	-	-	1	0	2	10	0	11	15	0	16	1	0	2	1	0	2	328	0	340
RF RX$_G$	32	2	27	7	3	3	4	2	2	12	6	6	15	7	7	4	2	2	-	-	-	1	1	1	4	1	3	11	4	6	1	1	1	1	1	1	95	32	61
LCD	7	0	8	50	0	51	25	0	26	2,029	0	2,030	759	0	760	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	2,908	0	2,919
LCD$_G$	1	1	1	7	3	3	4	2	2	135	39	39	103	50	50	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	270	104	109
Ethernet	74	0	75	65	0	66	25	0	26	33	0	34	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	1,189	0	1,201
Ethernet$_G$	32	2	26	7	3	3	4	2	2	4	2	3	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	524	21	395
Blink2	7	0	8	50	0	51	205	0	206	1,080	0	1,081	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	1,397	0	1,408
Blink2$_G$	1	1	1	7	3	3	64	31	31	72	36	36	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	168	82	87
Color	375	0	376	50	0	51	30	0	31	477	0	478	17	0	18	13	0	14	-	-	-	1	0	2	56	0	57	13	0	14	1	0	2	1	0	2	1,034	0	1,045
Color$_G$	11	2	9	7	3	3	4	2	2	55	18	19	4	2	2	4	2	2	-	-	-	1	1	1	47	23	23	10	3	6	1	1	1	1	1	1	145	58	69
K+ir	1,260	0	1,261	50	0	51	391	0	392	27	0	28	22	0	23	13	0	14	8	0	9	1	0	2	1	0	2	15	0	16	1	0	2	714	0	715	2,503	0	2,515
K+ir$_G$	122	11	105	7	3	3	126	62	62	58	29	29	4	2	2	4	2	2	3	2	2	1	1	1	1	1	1	10	3	6	1	1	1	704	352	352	1,041	469	566
lock	1,628	0	1,629	50	0	51	396	0	397	3,333	0	3,334	22	0	23	13	0	14	467	0	468	1	0	2	1	0	2	18	0	19	1	0	2	760	0	761	6,690	0	6,702
lock$_G$	151	11	111	7	3	3	126	62	62	723	354	354	4	2	2	4	2	2	329	38	132	1	1	1	1	1	1	12	4	6	1	1	1	751	375	375	2,110	854	1,050

PIO - Parallel Input/Outputs **UOTGHS** - USB OTG High Speed **TC** - Timer Counter **EEFC** - Enhanced Embedded Flash Controller
ADC - Analog-to-Digital Converter **PMC** - Power Management Controller **WDT** - Watchdog Timer

firmware. Given the added complexity of these graphs, it is reasonable to assume that some state transitions may no longer be as straightforward. To investigate exactly "how" the model was emulating these firmware, we kept track of every MMIO interaction and the effect that it had on the graph traversal. *Table* 4.3 enumerates the various non-standard transitions (*i.e.,* state transitions that did not have an immediately available edge from the current state). We define long jumps as a state transition that had to temporarily create a new transition (*i.e.,* the destination node was not reachable from the current node). The edge selection process is prioritized by locality and the number of edges that were merged to create the selected edge (*i.e.,* how many times that specific state transition was observed). Wildcards are edges that our algorithm deemed safe to accept *any* value (*e.g.,* the TX buffer in a UART controller). Finally, BFS transitions occur when the existing transition is not valid but a BFS through the graph was able to locate an acceptable edge. Fallback transitions were uncommon when emulating any of the initial firmware, as their recordings were used to generate the automata.

In *Table* 4.3, both `Blink2` and `Knock` observed multiple wildcard traversals due to their heavy usage of UART, which lends itself well to this. The 3,000+ long jumps in `Servo` are due to an interrupt handler accessing a memory address that was not available in the current state (*i.e.,* there was no sub-model for it). This is due to the fact that the emulated interrupts do not happen at the *exact* time that they were observed in in the recording. Nevertheless, the long jump selects a satisfactory node every time, and the execution continues correctly. This same phenomena occurred in the UART controller for `Blink2` and `Knock`, since the UART automaton is capable of supporting *any* of the interactions that were previously observed. Likewise for the multiple BFS traversals that were required.

4.4.4 Emulating Arbitrary Firmware

Finally, we exhibit Conware's ability to emulate a complete new firmware that was never seen in the training data. The hobbyist smart door lock program that we selected permits users to unlock the door by a knock pattern, a personal identification number (PIN) entered on the IR remote, or by presenting a specific color. In our recordings with the initial peripherals, we input sequences that would be accepted by the smart door firmware. However, these inputs could be replaced by a fuzzer, for example, in a straightforward way [18, 115, 116]. This particular firmware would require 13 wires to be connected to the Arduino and has 11 different physical peripherals, making it a non-trivial emulation target. Nevertheless, we were able to emulate the firmware our `Lock` model (*i.e.,* the automaton that was created from the individual recordings of the various peripherals using Arduino's included example code). Surprisingly, we observed *zero* failed reads or writes (*i.e.,* there were no reads or writes that our model was not able to handle). Verifying that our models worked "correctly" is not straightforward, since our goal is survivability of execution as opposed to a *perfect* representation. Thus, we first used high-level metrics, such as UART output, which this firmware had, and the distribution of MMIO accesses. In fact, the UART out was one-to-one identical as when we built and ran the real firmware.

To ensure that the peripherals were actually facilitating this interaction, and that our models were not just getting "lucky," we logged every MMIO accesses that the `Lock` firmware exercised and compared it to the peripheral recordings. Indeed, after aggregating *all* of the MMIO accesses from the recordings of each peripheral on the real hardware (using example code) and comparing the accesses to the execution of the `Lock` firmware we found that the same interrupts were fired, only five MMIO addresses (out of 94) were in the training data that were not observed in the recording. Indeed, these five addresses where all associated with the servo — specifically the PIOB controller and a PIO Pull Up Register. This makes sense, because the servo example code would increment the servo one degree at a time to move the motor slowly between *every* position, while the `Lock` only had an "on" and "off" position, which requires far less interaction. Moreover, we found that 348 of the address-value pairs were observed in the recording were also observed in emulation. In fact, 35 *new* unique address-value pairs where observed, while 375 pairs were never exercised by `Lock`.

Finally, we wanted to ensure that our peripherals were actually causing the firmware to execute most of its code (versus an error handler or a simple surface-level function). To measure this, we used QEMU's trace feature to record every basic block and function that was executed in the emulator. While the emulated firmware executed 738 unique basic blocks in the firmware, it was unclear if these were "interesting" basic blocks (*i.e.,* executing notable functionality). Thus, we used `angr` [137], a popular binary analysis platform, to identify every basic block (609 in total) and function entry point (68 in total) that was reachable from the `loop()` function (the main function in Arduino firmware). This list was then compared against the execution trace from QEMU, revealing that the emulation executed 362 (59%) of those basic blocks and 58 (85%) of the functions. Indeed, these were not superficial functions either. The maximum depth of the call stack originating from `loop()`, as discerned by `angr`, was six. The emulation results are shown in *Table* 4.5.

Table 4.5: Depth of call graph executed in emulation for `Lock` (the maximum possible is 6)

Depth of Function	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Number of Functions Executed	1	6	21	17	21	7	4

These measurements demonstrate that CONWARE is not only able to emulate firmware so that it survives, but that our models are successfully *coning* the firmware into executing its full functionality.

4.5 Discussion and Future Work

While this work makes many advances in peripheral emulation, there are still many avenues that we believe will provide interesting future research. First, it is likely possible to make even more concise automata by relaxing the comparison of `Pattern` sub-models to permit more generalized states. For example, it does not matter how many times `BUSY` is returned, but it does matter how many times a `1` is returned in an encoding scheme. Using re-enforcement learning or binary analysis to infer these cases could greatly improve modeling. Similarly, accurately correlating interrupts to the specific write that triggered them, versus the state transition, makes more sense for some peripherals (*e.g.,* UART). Indeed, we wrote a script to simply ($\approx$ 10 lines of Python) to *disable* buffering on our UART controller for debugging purposes. Methods for automatically detecting when these correlations do and do not hold will likely lead to more accurate emulation. Finally, while we do not see never-before-seen reads and writes in our evaluation, this scenario is inevitable. Employing a method that leverages static analysis to deduce appropriate reactions, and modifying the automata (*i.e.,* perform on-the-fly static analysis to construct a suitable response) appropriately sounds particularly fruitful.

4.6 Conclusion

The ability to emulate a system for analysis is critical for most security analyses, and yet this critical tool is completely absent form the world of embedded systems, despite importance of securing these prolific systems. In this work we present Conware, a system that is capable of automatically modeling hardware peripherals used by embedded systems and using these models to facilitate full-system emulation. Conware is a complete suite of software that facilitates recording peripheral interactions on real hardware, generating high-fidelity models from these recordings, and emulating firmware using popular emulation frameworks (*e.g.,* QEMU and Avatar^2). Conware's differentiator is that it is able to *merge* recordings in a pluggable way, enabling analysts to generate models based on one (or more firmware) recording and then use those models to execute a complete different firmware. This is critical to facilitate the emulation of firmware for systems that may not have debugging interfaces or that are executed on prohibitively expensive or elusive hardware. Conware was tested against various popular peripherals and was able to successfully emulate all of them. Moreover, we demonstrated Conware's ability to emulate a black-box firmware sample by merging six independent models, which were generated using the sample code that accompanies each peripheral, to create an emulation environment that was suitable for the new, never-before-seen, firmware.

Chapter 5

Protecting Embedded Systems from Physical Attacks

5.1 Introduction

Hardware-induced faults [138], which we refer to as *glitches*, are capable of corrupting the system state by modifying both instructions and data, and can be leveraged to undermine software-based security mechanisms, even if the software security mechanisms are implemented *with no semantic vulnerabilities*. Indeed, malicious glitches have been leveraged to compromise secure smartcards [31–33], security-hardened gaming consoles (*e.g.,* the XBOX 360 [34], Playstation 3 [35], Playstation Vita [36], and Nintendo Switch [37,38]), and enterprise IP phones [39]. Glitching attacks have even been leveraged to bypass both Intel's SGX protections [41] and ARM's TrustZone [42] and even extract hardware-embedded cryptographic keys [40]. However, little has been done to adequately study and defend against these types of attacks in practice. Some code-level glitching mitigations [139] have been proposed, but have not had their underlying assumptions or efficacy evaluated on real-world systems. Alternatively, custom-built hardware-based counter-measures (*e.g.,* brownout detection or lock-step computation) [140] are currently only sparsely deployed, due to cost and complexity, leaving the majority of embedded systems susceptible to glitching attacks.

Glitching attacks involve introducing a physical disturbance to a system that will ultimately corrupt the instructions being executed or the data being manipulated. This corruption can be achieved by changing the supply voltage [141, 142], optical probing with lasers [143,144], disrupting the clock [145], or introducing an electromagnetic pulse (EMP) [146,147]. To leverage these faults in a successful attack, the fault must be injected at a specific time in the execution pipeline. For example, if the execution was corrupted precisely when a security-critical branch condition was being checked (*e.g.,* checking the kernel's signature [148]), that instruction could be changed to a *no operation* instruction, and effectively skipped, allowing the attacker to disable secure boot [39, 149], escalate privileges [150], or extract "protected" code [36].

While effective defenses against other physical attacks are becoming commonplace in commodity computing systems (*e.g.,* trusted boot and encrypted memory), glitching defenses are still lacking. We hypothesize that this is likely due to a general lack of understanding about what exactly glitching attacks are capable of, and, subsequently, a systematic way to implement defenses against them. Indeed, we have observed a large disconnect between theory and practice in this field. For example, many researchers believe that glitching is capable of changing any pointer (*e.g.,* the program counter) in memory or making arbitrary code modifications because of published papers demonstrating this [150–152]. However, these effects are only realistic in laboratory environments with systems that are very-well understood and have already had the appropriate glitching parameters "tuned." For all intents and purposes these types of attacks are impossible in practice.

In this work, we introduce an open-source QEMU-based glitching emulation environment. This framework was used to exhaustively evaluate an ISA's instruction encoding against specific glitching effects (*e.g.,* bit flips), and examine the result of those instruction-level effects against a program's control flow. These flipped bits ultimately change the instruction being executed or the data being evaluated in a way that is beneficial to the attacker. In fact, our analysis confirmed that by simply flipping bits, the glitch can effectively "skip" an instruction with a high likelihood (*i.e.,* changing the targeted instruction into a *no operation*). We also found that this effect is often non-uniform. For example, on 16-bit ARM processors, glitches that tend to flip bits from 1 to zero appear to be exceptionally powerful (*i.e.,* "skipping" all branch instructions more than 60% of the time), while glitches that flip zeros to ones were less so (*i.e.,* "skipping" branch instructions less than 30% of the time).

In addition to emulating glitches, we also used a popular glitching tool (*i.e.,* the ChipWhisperer [153]) to conduct a suite of real-world glitching experiments to examine the effects of glitching on critical program elements (*i.e.,* control-flow-related instructions and data). In particular, our experiments were focused on using glitching to evade guard conditions. This evasion could be used to bypass security-critical code (*e.g.,* verifying signed code, disabling a debug interface, or checking user permissions). Our real-world glitching results provide new insights into how this corruption ultimately affects control flow. For example, load and store instructions appear to be more susceptible to glitching; the value being compared affects the *glitchability* of a branch condition (*e.g.,* `while(!a)` is more vulnerable than `while(a)`); and instructions which simply manipulate registers (*e.g.,* addition) appear to be exceptionally difficult to glitch. We leverage these findings to build our defense framework.

We present the first automated, open-source glitching defense framework, GlitchResistor, which is capable of adding various glitching defenses at compile time to *any* source code in an architecture-independent way. GlitchResistor implements numerous proposed glitching defenses (*e.g.,* double checking branches and loop guards, injecting random timing, and integrity checking on sensitive variables). We used GlitchResistor, combined with our ChipWhisperer-based glitching framework to evaluate the efficacy of these defenses in practice, examining their ability to thwart glitching, as well as the size and runtime overheads that each incurs. GlitchResistor was able to successfully defend against, and detect, every single-glitch attack that we attempted in our evaluation, necessitating a successful *multi-glitch* attack (*i.e.,* a glitch that affects multiple clock cycles) to evade the implemented defenses. Even so, GlitchResistor was able to reduce the success rate of our most powerful, multi-glitch attack to 0.263% in the worst case and 0.00306% in the best case, with detection rates of 79.2% and 99.7% respectively.

In summary, we make the following contributions:

- a comprehensive analysis of glitching attacks and their effects on control flow,
- a framework for emulating glitching attacks,
- a breadth of glitching experiments that characterize the effects of glitching and demonstrate the effectiveness of various software-only defenses,
- GlitchResistor, the first extensible glitching defense tool for automatically protecting vulnerable code, and
- an evaluation of GlitchResistor on real-world hardware, which demonstrates the effectiveness of software-only defenses, minimizing the likelihood of a successful attack and effectively detecting all glitching attempts in practice (`https://github.com/ucsb-seclab/glitch-resistor`).

5.2 Background

Fault injection is well-studied in the context of ensuring the reliability of a computer system [138]. Both software [154] and hardware [155] induced faults are capable of modifying the state of a system and disrupting its typical execution. Indeed, the act of inducing malicious software faults, which materialize as software bugs and vulnerabilities, has spawned an entire subfield of bug finding [156] and fuzzing techniques [157]. In contrast to software faults, malicious software-induced hardware faults were widely ignored by the software community until the relatively recent exposure of Spectre [158] and Meltdown [159] (microarchitecture attacks) and Rowhammer [160, 161], an attack against dynamic random access memory (DRAM). Malicious physical hardware-induced faults are still relatively unexplored.

Hardware-based attacks can be done either invasively (*e.g.,* decapsulating the chip [125]) or non-invasively (*e.g.,* through electromagnetic interference [162]). Non-invasive glitching techniques allow an attack to go undetected and typically permit the attacker to repeat the attack indefinitely. The general idea behind glitching is to interfere with the normal operation of a Flip-Flop circuit, transistor, or capacitor, to change the stored value or the output of an execution. This can be done using any form of interference, be it an external physical phenomena, like temperature or electromagnetic (EM) interference, or by operating the system outside its designed conditions (*e.g.,* by modifying the voltage or clock). In practice, voltage glitching, which is done by either increasing or decreasing the voltage for a brief period of time, and clock glitching, which involves inserting additional clock edges, are the most common glitching techniques, due to their relatively low cost and their effectiveness.

In this work, we only examine *non-invasive attacks*, as defenses against invasive attacks necessarily require hardware modifications. For those who are interested in the

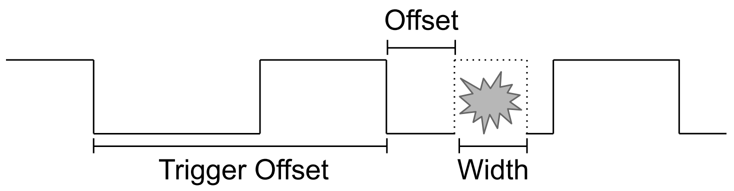


Figure 5.1: The three parameters that need to be tuned for clock glitching: the offset from the trigger, the offset into the clock cycle, and the width of the injected clock cycle

specific effects of each type of non-invasive glitch, we refer to reader to *Section* 5.3).

5.2.1 Motivation

Glitching attacks have already been used to attack numerous commercial systems. For example, researchers were able to use glitching to defeat the security on two automotive safety integrity level (ASIL)-D[1] compliant automotive microcontroller units (MCUs) [44], evading hardware-based countermeasures like Flash error-correcting code (ECC) and lockstep execution, using EM and voltage glitching, respectively. The same researchers were also able to bypass authentication checks, and even re-enable the JTAG interface. Similarly, voltage glitching has also been used to extract both Rivest, Shamir, and Adleman (RSA) [163, 164] and advanced encryption standard (AES) [40, 165] keys, and has even been shown to be effective against programs executing on modern Android phones and the Raspberry Pi, both running Linux [166]. More powerful attacks have even been able to control the program counter (PC) directly with glitching [151, 152]. In the case of defeating a secure boot loader, which has a relatively small attack surface and takes little or no user input, glitching attacks are one of the only methods for compromising the boot loader's security.

[1]The most stringent ASIL requirements of safety and fault tolerance.

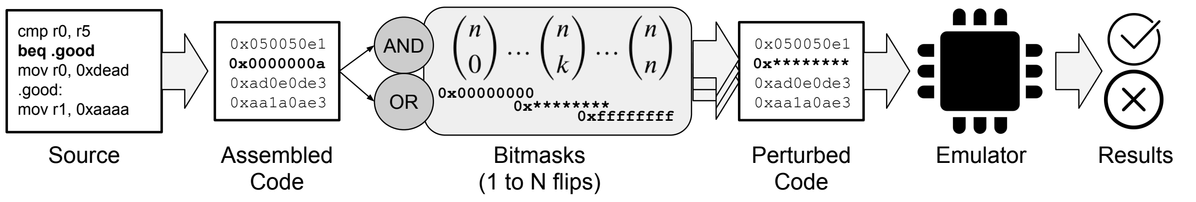


Figure 5.2: Depiction of our emulation framework for evaluating various corruption models (*e.g.,* OR and AND).

5.2.2 "Tuning" the Glitch

All glitching techniques necessarily require a "tuning" phase where the location and specific glitching parameters are tweaked until the desired effect is achieved. The attacker must first figure out *when* to inject the glitch, by calculating an offset from a known *trigger* (*i.e.,* an observable artifact that indicates which code is currently executing). For example, to inject a clock glitch, an attacker must simultaneously configure both the width and location in the clock cycle to inject a glitch, as well as the offset from an observable trigger (see *Figure* 5.1). Similar parameters must be tuned for both voltage and EM glitches (*e.g.,* the duration and voltage of the attack or the location and intensity of the EMP).

In our ideal laboratory environment with a perfect trigger, we were able to consistently, and automatically, tune our clock glitching parameters and successfully glitch an unprotected embedded system 100% of the time (10 out of 10 attempts) in less than 16 minutes, in the best case. However, this is only possible in practice with an initial search over the parameter space, which is the exact step that our evaluated defenses are targeting.

5.2.3 Defenses

Hardware-based defenses typically involve inserting additional circuits (*e.g.,* to detect voltage glitches [167]), an additional run-time monitor [168, 169], or CFI signatures [170, 171]. However, hardware modifications are impractical for the many already-deployed IoT devices. They are also far less likely to be adopted for individual systems, due to the lead times on hardware fabrication. Therefore, software-based techniques are more likely to be useful as practical defenses.

Software-based glitching defenses can never completely mitigate the problem. In the limit, glitching could (in theory) be used to skip *every* defensive instruction and even transform benign instructions into malicious ones. Nevertheless, software-based techniques are cheaper to implement and can be effective at defending against real-world attacks (in practice) by making the required scenario for a successful glitching attack increasingly improbable. Unfortunately, existing techniques, which rely on redundancy [172], only work on simple code-bases and have simplistic attacker models, which makes them infeasible on real-world code.

5.3 Glitching Effects

In this section we summarize and formalize the effects from each type of non-invasive glitching technique that have been reported in open literature (see *Table* 5.1). Recall that most processors, including embedded systems, use an execution pipeline to optimize their CPU usage. The most basic form consists of four stages, which each instruction passes through: *fetching* the instruction from memory, *decoding* the instruction so that it can be executed, actually *executing* the instruction, and finally storing the result in the *writeback* phase. A glitch could be injected during any one of these phases, and the result will vary depending on which stage was targeted. Moreover, full system

Table 5.1: Summary of the high-level effects that can be induced by various glitching techniques

		Glitch Type		
	Effect	**Voltage**	**Clock**	**EM**
Instructions	Corrupt individual instruction	✓	✓	✓
	Repeat previous instruction		✓	
	Corrupt, or repeat, instructions indefinitely		✓	
	Repeat previous n instructions, skipping n			✓
Data	Corrupt memory-related operation	✓	✓	✓
	Corrupt data in memory			✓

glitching techniques (*e.g.,* voltage glitching) can affect numerous instructions at once, but at different stages of the pipeline. For example, a single glitch on the system depicted in *Figure* 5.3 could, in theory, corrupt the write back of instruction one, the execution of instruction two, the decoding of instruction three, and the fetching of instruction four. In practice, voltage glitching appears to be more effective against the instruction fetch stage [173], whereas electromagnetic fault injection (EMFI) is more effective against the execution and writeback stages [174].

Voltage. Glitching the voltage of a system is one of the cheapest, most straightforward, and effective methods of glitching. It is capable of corrupting individual instructions [151, 163] (*i.e.,* transforming them into different instructions) and corrupting data that is being written to, or read from memory [163]. Likely due to the fact that voltage glitching is typically done by *decreasing* the voltage, others have observed that a voltage glitch typically flips 1s to 0s when it corrupts an instruction or data item [173,174]. Due to the system-wide nature of voltage glitching, has the potential to corrupt unrelated registers and instructions [173], and appears to be more successful on power-hungry operations (*e.g.,* memory loads or stores). However, voltage glitches by nature must be very brief

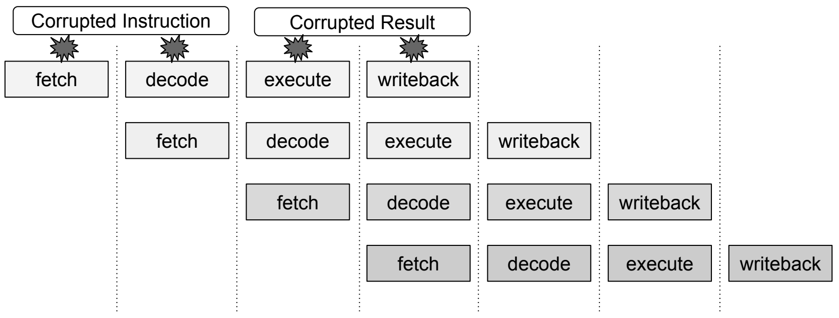


Figure 5.3: Example of a simple instruction pipeline with glitching effects called out for each stage

(*e.g.,* two rising clock edges [175]), as most chips have brownout detectors, or will simply turn off if the voltage is too low for too long. Similarly, supplying too much voltage can permanently damage the chip being glitched.

Clock. Glitching the clock is the most powerful non-invasive attack, as it can affect individual clock cycles, and thus inject very precise glitches. Previous work [175] has shown that clock glitching is capable of affecting every phase of the pipeline, with the following observed effects: invalid or zero result for arithmetic operations, invalid memory reads and writes (*i.e.,* the result containing zero, the address instead of the value, or a random value), and freezing the instruction buffer from being updated (*i.e.,* repeating the previous instruction). Because of the nature of the attack, an invalid clock edge could potentially be injected at *every* clock cycle, which means that this type of attack could be used to skip, or corrupt, an indeterminate number of instructions. Some have claimed the effects to be identical to voltage glitching [176], and they can, indeed, be combined to increase the overall success rate of an attack [175]. However, clock glitching does not appear to permit arbitrary corruptions [145]. Instead, the corruptions tend to be a mix of the previous instruction, or data, and the current, glitched, instruction (*e.g.,* 1100 $\leftrightarrow$ 0100 $\leftrightarrow$ 0000 $\leftrightarrow$ 0001 $\leftrightarrow$ 0011).

EM. EMFI attacks are the least invasive (requiring only a probe to be placed next to the chip) and yet are still effective in practice, with success rates of up to 96% [177] in a laboratory setting. The specific effects demonstrated in previous work [147,174] include: invalid arithmetic operations, corrupt instructions, and corrupt memory operations. The most powerful EM attack demonstrated was able to effectively freeze the instruction cache and repeat the previous four instructions, effectively skipping the next four instructions, while still incrementing the PC. When glitching during load and store operations (*i.e.,* fetch) with Flash memory on a Cortex-M3 processor, researchers were able to reliably flip bits from 0 to 1, where a higher voltage resulted in more bit flips [147] (they did not see any cases of 1s flipping to 0s). EMFI attacks can achieve the same effects as clock or voltage glitching (*e.g.,* corrupting a single instruction or data), but with more locality and without much collateral damage [146]. Similarly, when the execute and writeback stages were targeted, EMFI also appears to be able to flip 0s to 1s [174]. Finally, unlike other glitching attacks, EM interference is capable of changing the state of memory that is not currently being acted upon [39,161].

Temperature. Temperature glitching is done by operating a component outside of its temperature threshold, which can result in undocumented behavior. This technique can be used on its own to induce faults [178], but is likely more useful when coupled with other glitching techniques to increase their success rates [149,179].

5.4 Threat Model

Non-invasive glitching attacks require physical access to the device being glitched and control over the specific input being glitched (*e.g.,* the voltage line, clock line, or access to microchip). An attacker can dismantle any external packaging (*e.g.,* remove the case containing the electronic components), but cannot modify the electronic components in any non-reversible way. For example, an attacker may solder a wire to a specific pin to bypass a voltage regulator, but cannot remove or modify the integrated circuit (IC) directly.

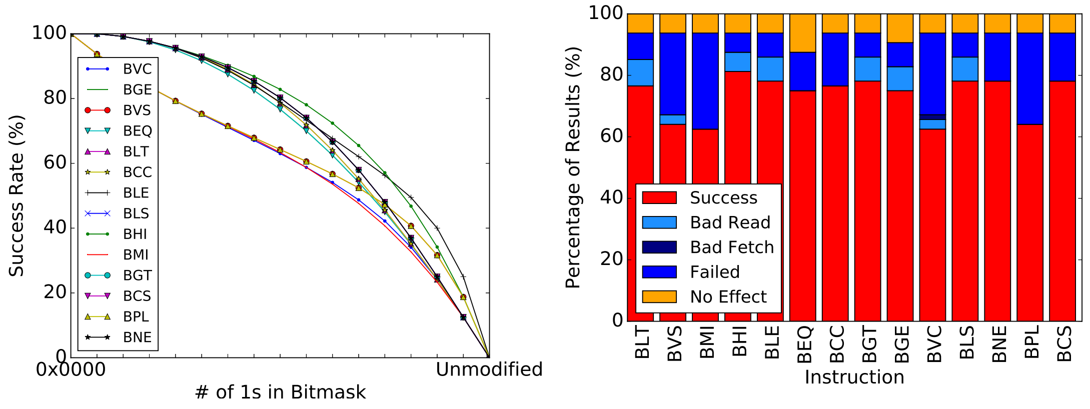
This threat model is realistic for any deployed embedded system: IoT devices, gaming systems, automobiles, robots, or military drones. The goal is typically to either bypass a secure check in the firmware or extract the firmware image for reverse engineering. As previously mentioned, the system must necessarily have some externally observable trigger to create a reliable glitch (*e.g.,* a voltage dip, an observable output, or a request for user input). In the various high-profile glitching attacks against gaming systems [34–38], the exploits were crafted by first identifying the *approximate* area that appeared to be vulnerable (*e.g.,* right before an error code) and then tuning the glitching parameters (*e.g.,* clock waveform, voltage modification, or EM power and position). No two systems are physically identical, which means that each attack must be specialized for the specific system being attacked. Even commercialized attacks (*e.g.,* the XBOX reset attack) are typically probabilistic, due to physical limitations, and have some method for automatically retrying the glitch in the event of a failure.

5.5 Glitching Effects in Emulation

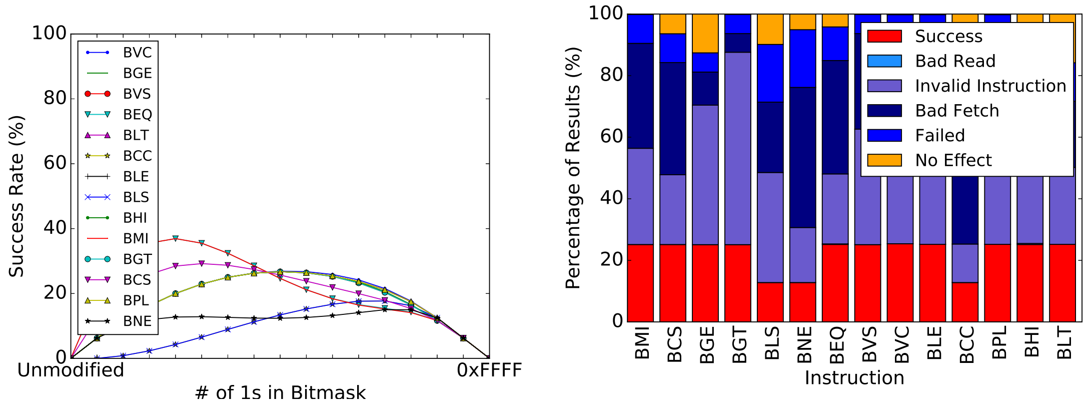
To gain a better understanding of how glitches affect the system, we first investigate the following research question:

RQ1 What is the likelihood that random bit flips will result in a "skipped" control-flow instruction?

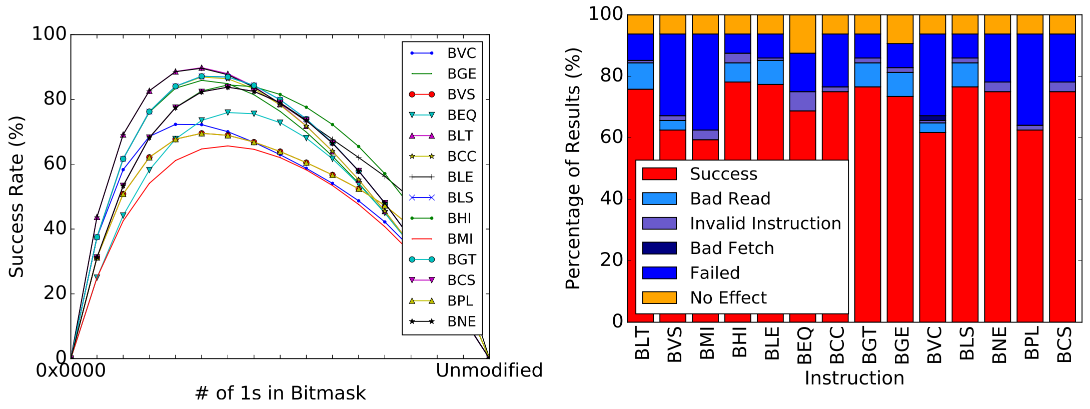
To quantify the effects of bit flips on a specific ISAs, we built an emulation framework that is capable of forcing bit flips (*i.e.,* corrupting specific instructions) and executing



(a) AND



(b) OR



(c) AND (`0x0000` Invalid)

Figure 5.4: The probability of a glitch succeeding on ARM Thumb as a function of the number of bits that were flipped and how they were flipped, *i.e.*, 1s to 0s (`AND`) or 0s to 1s (`OR`), computed by taking every possible combination, *i.e.*, $\binom{n}{k}$, of bits for each flip value and creating a bit mask that was either ANDed or ORed with the original instruction. The reasons for the failures are shown in the accompanying histograms.

the resulting code to determine the effects on the control flow of the program. Previous literature [145, 147, 151, 173, 175, 177] indicates that bit flips induced by glitching tend to be unidirectional (*i.e.,* either flipping 1s to 0s or 0s to 1s, but not both). While complex bit flips are possible, they are improbable in practice [151]. Therefore, we only consider unidirectional flips for our evaluation (*i.e.,* logical `and` and `or` operations).

We implemented our glitch emulator using Unicorn [180] for CPU emulation, Capstone [181] for disassembling code, and Keystone [182] for assembly. All of our test cases are manually written for the instruction in question such that a successful glitch (*i.e.,* the targeted instruction was skipped) will place the value `0xdead` in a known register, and a normal execution will place the value `0xaaaa` in a separate known register. Because these snippets of code are so small (*e.g.,* 3-5 lines of assembly), we are able to completely isolate the instruction in question. Our automated framework takes this source code, assembles it to machine code, and then generates *every* possible bit mask for *every* possible number of bits. More precisely, it produces $\binom{n}{k}$ possible bit masks for each k, where n is the number of bits in the instruction and k is the number of bits being mutated. These bit masks are then either `AND`ed or `OR`ed with the target instruction and then the entire program is executed in an emulator. Upon completion, the register values and error codes are read to log the result (*i.e.,* a successful glitch or the reason that it failed). This entire pipeline can be seen in $Figure$ 5.2.

We used this framework to quantify the effects of glitching on the popular 16 bit ARM Thumb architecture. While it can also be used to enumerate the effects of 32 bit architectures, we did not have access to the appropriate resources at the time of writing to enumerate these effects (the experiments are already constructed for both ARM and MIPS 32 bit and will be released with the source code). For example, a 32 bit exhaustive search would require $\sum_{k=0}^{32} \binom{32}{k} = 2^{32}$ emulations. The results for *every* conditional branch instruction in ARM Thumb under the `AND` and `OR` perturbation conditions can be seen in *Figure* 5.4. In these figures, a glitch is considered a "success" if the instruction immediately following the conditional branch, which would otherwise not be executed, was executed successfully. The failures are grouped in the following way: a *bad read* is when the system attempted to read unmapped memory; an *invalid instruction* is thrown when the emulator did not recognize the perturbed instruction; a *bad fetch* is thrown when an instruction was fetched from unmapped memory (*e.g.,* the PC was modified); an unknown *failure* is any unrecognized error; and, if the modification had *no effect* on the execution of the code, we annotate it as such.

One immediate observation is that the `AND` model exhibits a substantially higher success rate than the `OR` model. Initially we hypothesized that this was because in our experiments, the conditional branches had a relatively low hamming weight (*e.g.,* `beq #6` is repressed as `0b1101 0000 00000000` [183]), and thus converting them all to zeros, which is interpreted as `mov r0, r0` (or *no operation*) in ARM Thumb, was highly likely. However, after modifying our emulator to interpret all 0s as an invalid instruction, which is already the case for all 1s, this hypothesis was quickly debunked. The overall success rate for most of the bit masks were effectively unchanged (see *Figure* 5.4). Thus, it appears that the ISA itself is simply vulnerable to glitches that are capable of flipping 1s to 0s, which is also unfortunately the most likely effect of the cheaper, more popular, forms of glitching (*i.e.,* voltage and clock). Nevertheless, in practice, we hypothesize that this minor modification to the ISA could pay large dividends. Similarly, adding invalid instructions in between valid instructions would likely thwart many glitching attempts. However, the only way to test these hypotheses would be to fabricate a microchip with a modified ISA, which is out of scope for this work.

Table 5.2: The number of successful glitches for each clock cycle, mapped to the respective instruction that was executing and with a post-mortem view of the comparator register

Cycle	Instruction	Successes	`R3`	Count
0	`MOV R3, SP`	44	0	44
			8	32
			0x21	33
			0x68	1
1	`ADDS R3, #7`	9	8	8
			0xFF	1
2	`LDRB R3, [R3]`	-	-	-
3		18	0	18
4	`CMP R3, #0`	43	0	1
			8	37
			0x55	2
			0x20003FE8	3
5	`BEQ .loop`	89	8	41
			0x55	4
			0x20003FE8	44
6		133	8	49
			0x55	3
			0x20003FE8	73
			0x20003FEF	2
			0x28004309	6
7		183	0	41
			8	102
			0x20003FE8	36
			0x28004309	1
			0x40007FD7	1
			0xDFFFC010	1
			0xFFFFFFF9	1
	Total	553 (0.705%)	12 unique	

(a) `while(!a), R3=0x1000 initially`

Cycle	Instruction	Successes	`R3`	Count
0	`MOV R3, SP`	84	0	11
			1	38
			0x55	33
			0x68	1
			0xFF	1
1	`ADDS R3, #7`	14	0	4
			0x55	10
2	`LDRB R3, [R3]`	-	-	-
3		-	-	-
4	`CMP R3, #0`	-	-	-
5	`BNE .loop`	9	0	9
6		39	0x55	32
			0x20003FF6	1
7		126	0	4
			1	39
			8	1
			0x55	82
	Total	272 (0.347%)	7 unique	

(b) `while(a), R3=0x1000 initially`

Cycle	Instruction	Successes	`R2`	Count
0	`LDR R2,[SP,#0x10+a]`	25	0	1
			0x4EE6BB18	1
			0xE7D25763	23
1		-	-	-
2	`LDR R3,=0xD3B9AEC6`	-	-	-
3		1	0xE7D25763	1
4	`CMP R2, R3`	1	0xD3B9AEC6	1
5	`BNE .loop`	46	0xD3B9AEC6	1
			0xE7D25763	45
6		150	0x40	2
			0x400	2
			0xE7D25722	1
			0xE7D25763	145
7		129	0x40	1
			0x400	1
			0xE7D25763	127
	Total	352 (0.449%)	7 unique	

(c) `while(a!=0xD3B9AEC6), R2=0x48000028, R3=0x1000`

5.6 Real-world Glitching

To glean insights into real-world glitching effects, we employed the popular open-source ChipWhisperer Lite, a suite of hardware and software tools that enable glitching and side-channel analysis. In our experiments, we wanted to evaluate the *upper bound* of glitching effectiveness (*i.e.,* the best case scenario for an attacker, and the worst case scenario for the system being glitched). Therefore, we used the STM32F071RBT6, a 48 MHz ARM Cortex M0 chip with a 3-stage pipeline, as our target board, and drove the clock directly from the ChipWhisperer (*i.e.,* the most powerful glitching attack proposed by previous work). Similarly, we created a *perfect trigger* for each instruction sequence that we wanted to glitch. More precisely, our trigger would apply voltage to a general purpose input/output (GPIO) pin exactly 1 clock cycle before the targeted instruction, which permitted precise, reliable glitches to be injected. These conditions are *ideal* for an attacker and should provide a reasonable upper bound on the capabilities of glitching attacks. We investigate the following research questions:

RQ2 What is the upper bound of glitching effectiveness?

RQ3 Does the value being compared affect its *glitchability*?

RQ4 How are branches being "skipped" (*i.e.,* which instruction is being corrupted, and in which way)?

RQ5 How much more difficult is a multi-glitch (*i.e.,* a glitch that affects multiple instructions)?

5.6.1 Glitching Effects

In theory, the actual value being compared should affect the ability to glitch a certain branch. For example, glitching a 1 into a 0 *should* be easier than glitching $0b1010$ into

$0b0101$. To test this, we constructed three distinct experiments to evaluate the following expressions: `while(a)`, where `a=1`; `while(!a)`, where `a=0`; and `while(a!=0xD3B9AEC6)`, where `a=0xE7D25763`. These are all implemented as empty infinite loops, with `volatile` variables so they are not optimized out by the compiler (a successful glitch would *exit* the loop). The hypothesis being that `while(a)` and `while(!a)`, which are common in C code, should be much easier to glitch than values with a large hamming distance, as they both only require a single bit flip to change the outcome of the conditional branch.

To evaluate the effects of glitching on these three loops, we scanned *all* of the possible glitching parameters for each clock cycle in question. Each experiment, when compiled takes up to 8 clock cycles (the branch instruction can take between 1 and 3 clock cycles). Thus, we varied our clock-cycle offset between 0 and 7, and for each clock cycle ranged the width and offset of the glitch (*i.e.,* $[-49\%, 49\%] \times [-49\%, 49\%]$), resulting in 9,801 glitching attempts per clock cycle. The results of these three experiments, along with value observed in the comparison register, can be seen in *Table* 5.2.

Our results only partially corroborate our hypothesis, with `while (!a)` being the most vulnerable (0.705% success rate) and the other two achieving comparable success rates (0.347% and 0.449%, respectively). Surprisingly, the case where `a` was initialized to 1, and the condition was `while(a)` was the most resilient to glitching. However, when after examining exactly how the glitches were succeeding, a different story emerged. The assembly code for each case, along with the corresponding clock cycles, is also shown in each table. Since the processor being glitched has a three-stage pipeline, it is difficult to determine which instruction, and which portion of the pipeline was affected by the glitch, but the location of the glitch at least bounds the glitch's effects. For example, the the initial clock cycles (0 through 4), which set the values, appeared to be more susceptible to glitching in the simple comparison cases (*i.e.,* `!a` and `a`) than in the complex comparison case. This is likely attributed to the the fact that the underlying assembly instructions changed as a result of the comparison (*i.e.,* during the fetch stage). But the the fact the instructions have fewer glitchable clock cycles is still significant. In fact, the case for `while(!a)` by far had the most data corruptions that resulted in the branch condition being satisfied, as any non-zero value would suffice.

To explain some of the values that were observed in the resulting comparison register, we attached a JTAG debugger to the board and examined the state of the system *before* the loop was entered. For every case, `0x20003FE8` is the value of `SP`, `0x48000028` and is the GPIO address that was written to. Thus, `0x40007FD7` is likely a mix of the GPIO address and some corruption (*Table* 5.2). Similarly, for the `while(a)` case, `0x20003FF6` is likely a mix of `SP` and some corruption (*Table* 5.2). Interesting, in the `while(a!=0xD3B9AEC6)` case, 2 of the glitches resulted in the comparison register, `R2`, being correctly set to the unlikely value of `0xD3B9AEC6`, which is not on the stack, but is only stored as intermediate (*Table* 5.2). This must mean that the the `LDR` instruction was corrupted to load the valid into the wrong register. Similarly, the various the `0x4` values are likely a residual from the address in the register during a load. We were unable to identify any obvious connections to the other values stored in the registers, and can only assume that they are attributed to random bit flips.

5.6.2 Locating Optimal Parameters

We also investigated the *best case* scenario for glitching an unprotected conditional branch. In this experiment, we sought to identify glitch parameters that would have a 100% success rate. To achieve this, our algorithm starts by scanning our glitching parameters (*i.e.,* target offset, width, and offset) with a 10 cycle clock glitch, which encompasses every instruction in the `while` loop. Once successful parameters are identified, the algorithm then tests each individual clock cycle within the 10 clock-cycle range and recursively increases its precision (*i.e.,* $\frac{1}{10} * depth$) until a 100% success rate (10 out of 10 attempts) is achieved. In fact, this algorithm proved to be quite effective, locating the optimal parameters when attacking a `while(a)` loop in less than 59 minutes. Indeed, the algorithm achieved 7,031 successful glitches out of 36,869 in its search for when using `val != 0` as the comparator. When applied to a `while(a!=0xD3B9AEC6)` loop (*i.e.,* numbers with large hamming distance), the algorithm converged in 16 minutes with 901 successful glitches.

5.6.3 Multi-glitch Attacks

Previous work has proposed implementing redundant checks to thwart glitching, which is based on the assumption that successfully glitching multiple instructions is a significant technical barrier for attackers [150, 184]. Thus, we constructed an experiment to find the upper bound on the effectiveness of triggering an *identical* glitch twice in a row (*i.e.,* the ideal condition for an attacker as the same tuning parameters should work for both glitches). We used the same comparisons that we used in our single glitch scenarios, but now with the trigger being reset, triggered, and a second glitch inserted (*i.e.,* two identical loops back-to-back). We recorded the number of successful partial glitches (*i.e.,* the first glitch was successful but the second was not) as well successful multi-glitches (*i.e.,* both glitches worked and the execution skipped both branch conditions). The results from these experiments can be seen in $Table$ 5.3.

It is clear that multi-glitching is significantly more difficult in practice than a single glitch. The partial glitch success rates (*i.e.,* only the first glitch succeeded) are similar to those in our previous experiments: 1.330221%, 0.419600%, and 0.413223%, while the multi-glitch success rates (*i.e.,* the second glitch was also successful) were significantly lower: 0.493572%, 0.067595%, and 0.257627% respectively. Requiring a multi-glitch

Table 5.3: The number of successful partial and multi-glitch attacks against three different branch guards implemented as infinite `while` loops

	while(!a)		while(a)		while(a!=0xD3B9AEC6)	
Cycle	**Partial**	**Full**	**Partial**	**Full**	**Partial**	**Full**
0	77	12	83	24	23	7
1	20	2	19	-	2	-
2	2	-	1	-	-	-
3	124	87	-	-	-	-
4	326	211	1	-	-	-
5	166	36	30	2	47	36
6	161	17	49	2	136	99
7	167	22	146	25	116	60
Total	1043	387	329	53	324	202
Total (%)	1.330%	0.494%	0.420%	0.068%	0.413%	0.258%

reduced the probability of a successful glitch by factors of 6×, `while(!a)`, 3×, `while(a)`, and 1.6×, `while(a!=0xD3B9AEC6)`. While these results may seem higher that previous work would indicate, this experiment was construction to present the *best case* scenario for a multi-glitch. In practice, these factors would be significantly higher, since the attacker would not have 2 perfect triggers, the comparisons would likely not be identical, and the numerous physical limitations to generating multiple glitches in rapid succession become a factor (*e.g.,* voltage glitches and EMPs require some recharge time).

Potentially more interesting is the large gap between partial glitches and successful multi-glitches. This discrepancy leaves the potential to not only make glitching more difficult but to *detect* a glitching attempt, as a partial glitch introduces a logical impossibility, but do not entirely skip the instrumented checks.

5.6.4 Long Glitch Attacks

While the multi-glitch results are encouraging, clock glitching permits an even more powerful attack. Specifically, an attacker can inject a glitch at *every* clock cycle corrupting multiple contiguous instructions. Thus, we also tested the efficacy of a *long glitch* attack (*i.e.,* a glitch that is inserted for multiple clock cycles). In this experiment, we started by glitching 10 contiguous clock cycles (*i.e.,* the minimum number of clock cycles the two loops could possibly be completed in), and varied the clock cycles up to 20. For each number of repeated clock cycles, we varied the width and offset of the glitch in the same way as our previous experiments (resulting in 9,801 glitching attempts per clock cycle range).

Despite the potential power of this attack, we observed mixed results (see *Table* 5.4). The condition that was previously the most vulnerable, `while(!a)` faired much better against this attack, with far fewer successful glitches observed. We hypothesize that most successful glitching parameters, which disproportionately affect clock cycle 4 (*i.e.,* the compare instruction), are simultaneously corrupting the instructions before the comparator instructions and satisfying the exit condition. In the multi-glitch case the register would have contained 0, but in the long glitch case it is likely that the subsequent load was also glitched, disrupting the ideal conditions for the previously observed single-clock-cycle attacks. Conversely, the `while(a)` case appeared to be significantly *more* susceptible to long glitch attacks, with over a 10$\times$ increase in the success rate (*i.e.,* from 0.068% to 0.7%). We hypothesize that glitching so many load instructions could cause the various load instructions to fail, which would 0 in the register and satisfying the exit condition. The higher number of success between 10 and 12 cycle glitches appears to support this claim, as after 12 clock cycles the glitch would start to affect the compare and branch instructions of the second loop.

The lack of successes for the `while(a!=0xD3B9AEC6)` case coincides with our hypothesis that a glitch which simply changes the value in the register is unlikely to succeed. It appears that successful glitches against case are corrupting the comparison, the branch, or the actual value loaded. In a multi-glitch scenario, the targeted glitch was affecting

Table 5.4: The number of successful long glitches against three unique branch guards implemented as two subsequent `while` loops, obtained by attempting all glitch offsets, widths, and number of clock cycles using a powerful clock glitch

Cycles	while(!a)	while(a)	while(a!=0xD3B9AEC6)
0-10	20	96	35
0-11	19	140	20
0-12	6	92	8
0-13	7	55	6
0-14	9	66	8
0-15	6	74	7
0-16	6	54	4
0-17	7	62	4
0-18	9	50	6
0-19	9	46	5
0-20	11	52	4
Total	109	787	107
Total (%)	0.101%	0.730%	0.0992%

the same clock cycle both times, against identical code. However, in the long glitch case, there are other instructions in the way that will also get glitched, making it exceedingly unlikely *both* of the compare and branch instructions will be bypassed.

5.7 Glitching Defenses

While many glitching defenses have been proposed, few have been implemented, and we are unaware of any tool for generally applying these techniques. Thus, we present GlitchResistor, the first automated, open-source tool for implementing glitching defenses. GlitchResistor was implemented using the LLVM Project to modify both the source and compiled code (Clang and LLVM respectively). This enables GlitchResistor to support multiple architectures with relatively low overhead. Indeed, many of the defenses *must* be implemented as a compiler pass, since implementing them in source code would result in the compiler optimizing them away (*i.e.,* because they appear as logically impossible or dead code). In this work, we only focused on the ARM architecture, specifically the STM32 microcontroller, due to its proliferation in embedded systems, its development support, and the supporting glitching frameworks [153]. However, our defenses work, without modification, on any architecture that is supported by LLVM (*e.g.,* MIPS, PowerPC, and RISC-V architecture)

In general, software-based glitching defenses can be categorized into three broad categories: constant diversification, redundancy, and random timing.

5.7.1 Constant Diversification

Ideally, GlitchResistor would ensure that the set of enumerations (ENUMs) and return values would have a maximum minimum pairwise hamming distance (*i.e.,* the minimum hamming distance between *all* of the values would be maximized) to minimize the chance of bit flips modifying a value into a different valid value. However, this is unfortunately an open coding theory problem in the general case, *i.e.,* $A(n, d)$ [185]. Thus, GlitchResistor instead leverages Reed-Solomon codes to generate values with large pairwise hamming distances. In theory, this implementation can generate codes such that the minimum pairwise hamming distance is $b - \lceil log_2(c) \rceil$ where b is the size of the value in bits and c is the number of values being generated. However, we used a more general purpose open-source implementation [186], which provides a flexible, fast computation of Reed-Solomon error codes. Our current implementation is configured with a message size of two bytes (*i.e.,* up to 2^{16} unique values in a set) and an ECC length equal to the size of values being generate (*e.g.,* 4 bytes for a typical ENUM). GlitchResistor then generates a message for each number $[1, count]$ and uses the generated ECC as the new value in the program code. Ensuring a minimum pairwise hamming distance of 8.

ENUM Rewriter The ENUM Rewriter is the only defense implemented as a clang source code rewriter tool. This is because in the LLVM IR, used by a compiler pass, ENUMs will be replaced by corresponding constant values, and it is hard to detect which constant is the result of an ENUM expansion. Consequently, it is hard to replace ENUMs using a compiler pass in a sound manner. GlitchResistor first parses the abstract syntax tree (AST) of all the source and header files to identify ENUM declarations that have *all* of their values uninitialized. Then, for each of the uninitialized ENUM declarations, a set of Reed-Solomon codes is generated, and used as the declarations. GlitchResistor does not modify partially or fully initialized ENUM declarations, as they could represent certain expected values, and changing the values might affect the functionality of the firmware.

Non-trivial Return Codes GlitchResistor finds all of the functions that *only* return constant values using the LLVM *ModulePass*. For such functions, GlitchResistor examines how the return values are used by its callers. When they are *exclusively* used directly in branches (*i.e.,* compared to a constant) GlitchResistor replaces the return value and the constant that it is compared to with the hard-to-glitch values from our Reed-Solomon implementation. Our decision to only instrument functions that return constants reflects the fundamental difficulty in calculating all of the possible computed return values. Instrumentation that deals with such corner cases would be significantly more intrusive, and likely unsound. Our decision to only instrument return values that are used directly in branches could be relaxed, though only to a certain extent. If the instrumented constant is stored in an aliased memory location, a significantly more heavyweight instrumentation would be required to dynamically track the value and update all of the references appropriately. Despite these minor limitations, our return code protection instruments a reasonable number of functions in practice (*i.e.,* 24 out of 312 *total* functions in our evaluated firmware).

5.7.2 Redundancy

GlitchResistor's redundancy defenses are implemented as an LLVM compiler pass that replicates existing code to ensure that no single-glitch attack will be capable of corrupting the execution. We ensure that code added for redundancy is *not* optimized out by other compiler passes by marking the inserted `load` and `store` instructions as volatile. These checks are capable of detecting glitches, as the injected check will never be false under normal operating conditions. Others have proposed and tested simple instruction duplication [187], concluding that instruction duplication alone is likely not a cure-all solution; hence the multi-pronged approach.

Data Integrity GlitchResistor's data integrity protection is implemented by performing a *ModulePass*, which locates any global variables that were marked as *sensitive* by the developer (*e.g.,* by listing them in a configuration file). Once identified, these sensitive variables are replicated, and a second variable, which is used for verification, is allocated in a separate region of memory to ensure that it is not physically co-located with the initial variable. When a sensitive variable is written to memory, it is inverted (*i.e.,* `xor`ed with $\neg 0$ of the appropriate size), and this integrity value is stored in the complementary integrity variable. Then, when the value is later read from memory, both the original variable and the integrity value are read from memory and the operation will continue *if and only if* $var \oplus varIntegrity == \neg 0$, otherwise a glitch detection function will be called.

Branches and Loops GlitchResistor implements two *FunctionPass* transformations to replicate conditional branch conditions. The first replicates the *true* condition for *every* conditional branch in the control-flow graph (CFG). When replicating the branch condition, GlitchResistor also replicates any instructions that are needed to calculate the comparison (*e.g.,* loading a value from memory, mutating it, and comparing it to an immediate). However, not every instruction can be replicated. For example, volatile variables, function calls, and LLVM `PHINodes` cannot be replicated because they may have adverse side-effects, or are likely to change between checks. This redundant comparison is computed to be the opposite of the initial branch condition (*e.g.,* `if (a == 5)` would become if `if (¬a == ¬5)`), which ensures that the same bit flips repeated twice would not be able to bypass both checks. This defense assumes that security-critical operations are typically guarded by a conditional branch and that the default, *false*, branch is not as important to protect, as it will be taken most of the time. However, this assumption is does not hold with loops. Thus, GlitchResistor performs a second pass to add the same redundant instrumentation to the *false* branch of loop guards.

Detection Reaction GlitchResistor does not dictate an action to be taken when a glitch is detected, but instead provides a function that is trivially implemented by the developer. In fact, the specific reaction to a detected glitching attempt is *necessarily* application specific. For example, on a gaming system it may be sufficient to simply report the attempt or disable updates, whereas a critical military system may want to react more assertively (*e.g.,* completely destroying the data or device).

Random Timing

GlitchResistor currently injects randomness in the execution by injecting a random busy loop at the end of each basic block the code. The current implementation is a simple linear congruential generator (LCG) with the input parameters used by `glibc`, and each invocation executes between 0 and 10 no-operation (NOP) instructions. To ensure that any observable trigger is necessarily before the random function, the delay function is injected at the end of every basic block that ends in a `SwitchInst` or

`BranchInst` (*i.e.,* right before a branch). This code injection was implemented as an LLVM *FunctionPass*. Functions can be easily omitted when the module is configured in *opt-out* mode or included when it is configured in *opt-in* mode. Our seed is incremented, and written to flash, during the first invocation of the function (on our STM32 board, this was implemented in 10 lines of portable `C` code). GlitchResistor modifies the state of the random function immediately after the board boots (even before the board initializes) and writes the new seed to non-volatile memory to thwart repeated attempts against the same seed. This initialization code is also instrumented by the other defenses, which are capable of detecting failed glitching attempts.

5.8 Evaluation of Defenses

GlitchResistor was both developed and evaluated on real hardware. Specifically, we leveraged the STM32 suite of embedded devices. Two research questions arise with respect to defenses:

RQ6 How much overhead, in terms of both size and runtime, is incurred when using each GlitchResistor defense?

RQ7 How effective are the various GlitchResistor defenses at both mitigating and detecting glitching attacks?

5.8.1 Overhead

To evaluate the overhead imposed by GlitchResistor we first built a simple, indicative firmware using the STM32CubeMX code generator. This firmware initializes the board, and then loops forever, reading the number of ticks (*i.e.,* milliseconds) since the the board was booted and printing out performance information after every loop

Table 5.5: Time overhead imposed by each defense on the boot time of a standard STM32 firmware image(clock cycles)

Defense	Clock Cycles (Avg.)	% Increase	Constant	% Adjusted
None	1736	0.00%	0	0.00%
Branches	1933	11.35%	0	11.35%
Delay	184388	10521.45%	177849	276.69%
Integrity	1737	0.06%	0	0.06%
Loops	1737	0.06%	0	0.06%
Returns	1739	0.17%	0	0.17%
All\Delay	2082	19.93%	0	19.93%
All	184761	10542.93%	177993	289.88%

iteration using the UART interface. The variable that is used to store the tick counter was marked as a sensitive variable, and two functions which use ENUMs and constant return values are used to check the tick value. The firmware will call a *success* function if the tick value is ever equal to 0, which was designed to be impossible.

The specific board that we used in this experiment was an STM Nulceo 64 with an ARM Cortex-M4 (STM32F303RE)[2]. The default project, configured to be built with a `Makefile` was easily augmented to be built with LLVM and the appropriate GLITCHRESISTOR modules using a simple patching script that is provided with GLITCHRESISTOR. To ensure that there was no bias in the evaluation, we only measure the boot time of the system, as this code was provided by the CubeMX suite, and is used in numerous real systems. Moreover, the most security-critical code on embedded systems (*i.e.,* when GLITCHRESISTOR would provide the most value) is typically the bootloader. Each firmware was built using the default `-Og` optimization, which provides a *worst case* analysis. Furthermore, we can use existing static analysis techniques [188, 189] to further reduce the regions of code that need to be instrumented.

[2]This is different from our glitching examples, because this board is more readily available and requires no special hardware to test with.

Runtime

To evaluate the boot process in a chip-agnostic way, we use the number of CPU cycles as our metric for comparison. This was done by enabling the data watchpoint and trace unit (DWT) on the board, and then reading the CPU once when the board is reset, and again after the HAL and board had completely initialized. Since our board is doing relatively simple operations, it only takes 1,736 clock cycles to boot in the uninstrumented case. We evaluated each defense independently, as they can be used *à la carte*. The results are shown in *Table* 5.5.

Injecting delays incurs in a substantial constant overhead, as it must both read and write from flash memory the first time that it is called to update the seed to ensure that the pseudo-random number generator (PRNG) is unpredictable at every boot. When this constant overhead is accounted for, instrumenting *every* basic block in the boot process incurs a 277% overhead. However, in practice, a developer may want to use this particular feature in an "opt-in" way, such that it will only be applied to annotated functions. Without the delay defenses enabled on every basic block, the runtime overhead incurred, in terms of clock cycles, is less than 20%. Nevertheless, both of these overheads are likely acceptable in practice to protect the critical code regions in a deployed embedded system.

Size

Since most embedded systems have strict constraints on their size, weight, and power (SWaP), we also enumerate how much additional code is inserted by GlitchResistor. *Table* 5.6 depicts the various code segments that are affected by each defense in GlitchResistor. Again, injecting a delay into *every* basic block incurs the largest overhead (13%). Meanwhile, the other defenses only combine for a 15% increase in size, most of which is in the `.text` segment. While modifying constant values (*i.e.,* returns

Table 5.6: Size overhead imposed by each individual defense on a standard STM32 firmware built using CubeMX (bytes)

Defense	**text**	**text (%)**	**data**	**data (%)**	**bss**	**bss (%)**	**total**	**total (%)**
None	6456		120		1728		8304	
Branches	6956	7.74%	120	0.00%	1728	0.00%	8804	6.02%
Delay	7512	16.36%	128	6.67%	1768	2.31%	9408	13.29%
Integrity	6840	5.95%	124	3.33%	1732	0.23%	8696	4.72%
Loops	6840	5.95%	124	3.33%	1732	0.23%	8696	4.72%
Returns	6460	0.06%	120	0.00%	1728	0.00%	8308	0.05%
All\Delay	7700	19.27%	124	3.33%	1732	0.23%	9556	15.08%
All	9144	41.64%	132	10.00%	1768	2.31%	11044	33.00%

and ENUMs) should in theory be "free," we actually see that they increase the size of the binary slightly because the transformed values are all necessarily four bytes, while smaller values can be encoded as a single byte. While these overheads may seem large after an initial glance, it is a small price to pay for the protection provided.

5.8.2 Effectiveness of Defenses

When testing these defenses against real glitches, we created both a *worst case* and *best case* scenario. In both cases we marked our variables as `volatile`, which hinders the effectiveness of the defenses (*i.e.,* they should perform *better* in practice). This decision should provide a reasonable lower-bound for the effectiveness of these defense in practice (*i.e.,* their ability to protect *any* code). Similarly in both experiments, we attempted three different attack scenarios: a single glitch attack, where the clock cycle being glitched was varied (between 0 and 10); a long glitch attack, where the number of clock cycles being glitch was varied (between 10 and 100 at increments of 10); and a windowed long glitch attack, where the number of clock cycles was fixed at 10 (the best case in our previous experiment), but the initial clock cycle was varied (between 0 and 10 at increments of 10). All of the experiments had a *perfect* trigger, as before. These attacks are *far more powerful* than what an attacker would have access to on a real system, but, again, were constructed to provide a lower bound for the efficacy of the defenses.

`while(!a)` (worst case)

The `while(!a)` condition was the most vulnerable against single-glitch attacks, and was thus chosen as our *worst case* scenario. As in *Section* 5.6, we glitched the infinite loop, attempting to break out of it with the various defenses compiled into the code. While it should be theoretically impossible to defeat these defenses with a single glitch, the `volatile` variable leaves the possibility of successful glitching the register value and satisfy both conditional branches. The results from the three glitching attacks against this code are shown in *Table* 5.7.

The defenses turned out to be highly effective against the single-glitch attack, with success rates plummeting to less than 0.01%. Moreover, the detection rate is remarkably high (over 98%) both with and without the randomization defense enabled. This result is somewhat unsurprising, as these defenses were specifically constructed to ensure that no single incorrect branch would result in a compromised system [190]. However, the detection rates are especially encouraging with respect to real-world use cases for these defenses. The results are similarly positive against the more powerful long glitch attacks, with all of the defenses touting detection rates above 79%. It appears that the 10 cycle, windowed glitch is far more effective against systems that do not have randomization enabled, since the more targeted window produces fewer detectable side effects. However, with randomization enabled, this attack performs slightly worse that the *longer* long glitch attack, likely due to the fact this shorter glitch window is more likely to corrupt a branch condition in the random function, which would be detected. On the contrary, since the long glitch attack will affect *every* clock cycle after the trigger, it has the possibility of glitching *all* of the detection code that it may touch.

Table 5.7: Successful glitches and detections against an infinite loop and a branch condition with GlitchResistor defenses

		`while(!a)`		`if(a==SUCCESS)`	
		All	All\Delay		
Single	Total Glitches	107,811		107,811	
	Successes	10 (0.00680%)	4 (0.00408%)	-	1 (0.000928%)
	Detections	653 (98.4%)	1,032 (99.6%)	351 (100%)	95 (95.4%)
Long	Total Glitches	98,010		98,010	
	Successes	258 (0.263%)	262 (0.267%)	3 (0.00306%)	44 (0.0449%)
	Detections	981 (79.2%)	649 (71.2%)	1,143 (99.7%)	274 (86.2%)
10 Cycles	Total Glitches	107,811		107,811	
	Successes	227 (0.211%)	1,281 (1.188%)	10 (0.00557%)	2 (0.00186)
	Detections	1,858 (89.1%)	992 (43.6%)	2,019 (99.7%)	1016 (99.8%)

`if(a == SUCCESS)` (best case)

In real code, infinite while loops are unlikely to guard security-critical code. Thus, to provide a more fair evaluation of the proposed defenses, we also attempted the three attacks against a simple `if` statement that is more indicative of how programmers write code. To ensure that all of the proposed defenses would be used, and to use the most resilient branch condition from *Section* 5.6.3, we created an uninitialized `enum` variable: `SUCCESS`, which was initialized to `enum FAILURE`. This scenario should be the *best case* for the defenses (modulo the `volatile` variable), as the window for the a successful glitch is now quite narrow (*i.e.,* 8 clock cycles). The same attacks that from *Section* 5.8.2 where used against this `if` statement; the results are shown in *Table* 5.7.

Indeed, the real power of these software-only defenses is exhibited in this case — only *one* single glitch attack was successful, with detection rates above (95%). The effectiveness of the long glitch attacks were similarly diminished. With all of the defenses enabled, the best attack was only able to achieve a 0.00557% success rate, with over

2,000 detections (a 99.7%) detection rate. Without the randomization defense enabled, the best attack was able to achieve a success rate of 0.0449%, with an 86.2% detection rate. While this experiment was constructed to be the best case scenario for the defenses, it is certainly not a corner case in real world code, demonstrating some real promise for these types of software-only defense against glitching in practice.

5.9 Conclusion

In this work, we attempted to differentiate the theory and practice of hardware glitching. To this end, we created an emulation framework, capable of evaluating a given ISA's susceptibility to glitching, we examined the efficacy of various glitching attacks on real hardware, and we presented GlitchResistor, an automated, open-source software-only defense framework. Our emulated experiments confirm that bit-level corruption can "skip" control flow instructions in ARM with a high likelihood *in theory* (60% when flipping to 0 and 30% when flipping to 1). Our real-world experiments demonstrated that glitching can be highly effective when all of the variables are controlled (*e.g.,* 100% success rate), and that the values being compared affect the *glitchability* of a particular branch instruction (*e.g.,* `while(!a)` was 2× more susceptible to glitching than `while(a)`). Moreover, we provide insights into *how* the control flow instructions are being skipped (*e.g.,* the register data being corrupted *versus* the execution being corrupted). We also demonstrate the complexity involved with multi-glitch attacks, whose difficultly is the basis of many proposed defenses. Finally, we show that GlitchResistor, with it's various software-only glitching defenses are capable of completely eliminating single-glitch attacks *in practice* and can minimize the likelihood of a successful multi-glitch attack (*e.g.,* by bringing the success rate down to 0.000306%), while detecting failed glitching attempts at a high rate (*e.g.,* between 79.2% and 100%).

Chapter 6

Related Work

6.1 LO-PHI

VAMPiRE [191] is a software breakpoint framework running within the operating system. It runs in kernel mode, meaning it is safe for debugging ring 3 (user mode) malware. Rootkits can gain kernel-level privileges to circumvent VAMPiRE. However, as LO-PHI does not rely on the operating system, it can be used to safely debug rootkits.

Ether [59] is a malware analysis framework based on hardware virtualization extensions (*e.g.,* Intel VT). It runs outside of the guest operating systems, *i.e.,* in the hypervisor, by relying on underlying hardware features. BitBlaze [4] and Anubis [5] are QEMU-based malware analysis systems. They focus on understanding malware behavior, instead of achieving better transparency. V2E [60] combines both hardware virtualization and software emulation. HyperDbg [61] uses the hardware virtualization that allows the late launching of VMX modes to install a virtual machine monitor, and run the analysis code in the VMX root mode. SPIDER [62] uses Extended Page Tables to implement invisible breakpoints and hardware virtualization to hide its side-effects. Compared to our system, Ether, BitBlaze, Anubis, V2E, HyperDbg and SPIDER all rely on easily detected emulation or virtualization technology [52, 76–78] and make the assumption that virtualization or emulation is transparent from guest-OSes. In contrast, LO-PHI provides memory access directly from the PCI bus, greatly reducing the potential attack surface. In addition, traditional debugging techniques often add varying degrees of execution overhead. LO-PHI employs specialized hardware that is fast enough to decrease visible timing artifacts otherwise introduced by emulation.

BareBox [15] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation techniques. However, it only targets the analysis of user-mode malware, while LO-PHI can be used for debugging hypervisor rootkits and kernel-mode device drivers. BareCloud [16] is more similar to our approach, as it utilizes mostly un-instrumented machines and executes the binaries with a small software-based loader. Nevertheless, BareCloud requires a network-based storage device and only has information about the disk state before and after execution of the binary, whereas we are able to reconstruction the entire stream of file-system operations. Futhermore, BareCloud has no memory instrumentation and presents numerous detectable artifacts (*e.g.,* malware loader software, networked-drive). Willems *et al.* [93] used branch tracing to record all the branches taken by a program execution. As pointed out in the paper, the data obtainable by branch tracing is rather coarse, and this approach still suffers from a CPU register attack against branch tracing settings. However, LO-PHI provides fine-grained memory access over the PCI bus, and is thus resistant to CPU register mutation.

Virt-ICE [192] is a remote debugging framework. It leverages virtualization technology to debug malware in a VM and communicates with a debugging client over a TCP connection. However, since it uses a VM, a malware may refuse to unpack itself in the VM. LO-PHI accesses the raw host memory very rapidly, so we can transparently detect when this type of execution occurs.

There is a vast array of popular debugging tools. For instance, IDA Pro [193] and

OllyDbg [194] are popular debuggers running within the operating system that focus on ring 3 malware. DynamoRIO [195] is a process virtualization system implemented using software code cache techniques. It executes on top of the OS and allows users to build customized dynamic instrumentation tools. Similar to LO-PHI, WinDbg [196] uses a remote machine to connect to the target machine using serial or network communications. However, these options require special booting configuration or software running within the operating system, which is easily detected by malware. LO-PHI requires a PCI slot, but is intended to run on out-of-the-box consumer hardware, where debugging facilities may not be desirable.

6.2 Conware

The handling of peripheral interactions is one of the linchpins of the dynamic analysis for embedded firmware.

Initial dynamic analysis techniques leveraged hardware-in-the-loop analysis, where all of the interactions with the peripherals are forwarded to the real device. Avatar [20] was the first such emulation framework. Similarly, Charm [197] targeted smartphone drivers. However, Charm is designed for kernel drivers rather than arbitrary peripherals. Prospect [198] forwards peripheral accesses at the `syscall` layer. However, the `syscall` interface does not exist in most of the bare-metal firmware.

Several optimizations have been made over naïvely forwarding all of the peripheral accesses to the hardware. Surrogates [17] significantly improves the forwarding performance of Avatar via customized hardware. Kammerstetter *et al.* [199] uses cached peripheral accesses to minimize the interaction with real hardware. And Avatar^2 [118] generalized caching by allowing the replay of forwarded peripheral input and output without using the hardware.

Recently, several works proposed domain-specific models to handle peripherals. Here, effective peripherals models are carefully engineered, mostly manually, for a specific set of firmware. HALucinator [116] uses a model of a known HAL implementation to emulate embedded systems with the HAL. Similarly, PartEmu [200] and Exvivo [201] create peripheral models that can handle ARM-based, Trusted operating systems, and Android kernel drivers, respectively.

Few works try to handle peripheral interactions in a generic manner automatically. Pretender [115] is a record-and-model approach that first records peripheral accesses, generates access models, and then tries to intelligently reply. However, Pretender needs debug access to the chip being modeled and only supports simplistic peripheral models. P^2IM is a fuzzing-based approach that generates *acceptable* inputs by randomly fuzzing the firmware [119]. P^2IM only considers on-chip peripherals and requires their abstract models to be generated manually and offline by a domain expert. On the *con*trary, CONWARE can generate these models automatically can support both on- and off-chip peripherals.

6.3 GlitchResistor

Previous work [172], proposed and evaluated two software-only defenses (one which replicates instructions for redundancy [202] and the other which detects glitches [203]) for `bl` and `ldr` on ARM systems against EMFI glitching. These defenses are quite similar to our techniques for redundancy, and had similar success when they were evaluated. However, they noted that the countermeasure needed to be extended to a larger set of instructions and architectures, which GLITCHRESISTOR does by leveraging LLVM. Recent work [184] independently implemented, and evaluated branch duplication techniques in the context of spurious bit flips due to hardware malfunctions. Similar work [204] proposed a CFI method which implements a counter to detect if two more C source lines have been "skipped." However, this defense is especially heavyweight since it injects code after every instruction and it does not account for the possibility of a multi-glitch. Another work, CAMFAS [205], that uses SIMD [206] to replicate almost all instructions to detect fault attacks also suffers from the problem of being cumbersome and requires special hardware. Our LLVM modifications are similar to those implemented by Obfuscator-LLVM [207]. However, Obfuscator-LLVM is intended to defend against software faults and reverse engineering, whereas GlitchResistor is explicitly defending against hardware-induced faults.

Others have proposed a hybrid software and hardware approach where functions can be protected by inserting an `assert` function in the source, which will be updated with an LLVM pass to confer with the hardware and verify the "signature" of the function being executed [171]. GlitchResistor is differentiated by its fully-automated instrumentation and lack of source-code annotations.

Emulated glitching attacks has also been done previously. For example, one system implemented a fault injection emulator in the context of writing fault-tolerant code, but did not examine malicious glitching attacks [208]. Others similarly implemented a QEMU-based fault injection emulator [209] and glitch simulator [210] have been created to evaluate fault-tolerant techniques, both of which achieved mixed results.

Previous work [139] presented a comprehensive suggestions for source code modifications to make code glitch resistant, which our defenses are based on. Similarly, more recent work [44] advocated that "software mitigations like execution flow control, redundancy or random delays should be implemented" in embedded firmware. However, GlitchResistor is the first open-source framework for experimenting with various defenses and to test these defenses against attacks on *real hardware*, grounding our results and providing a more realistic view of their practical efficacy.

Chapter 7

Future Research Directions

I created and presented multiple novel techniques for analyzing embedded systems and insights into potential defenses. However, this research has subsequently presented new hypotheses and potential future research directions, which I will briefly enumerate here.

7.1 Continuous Introspection

LO-PHI demonstrated the practicality of using hardware introspection as a technique for performing specific analysis (*e.g.,* detecting rootkits) and the ability to debug firmware that would otherwise be prohibitively difficult due to cost or technical limitations. Yet, the idea of using hardware-based introspection as a continuous integrity monitor is still mostly unexplored. Hardware instrumentation offers an unabated view of the system that is difficult, if not impossible, for any software-based exploit to avoid, making it an ideal method for detecting software compromises in a way that cannot be subverted. For the full value of a hardware-based integrity checking technique to be realized, it would ideally be fully automated and offer the ability to automatically *heal* the software. This would require a novel technique for automatically bridging the *semantic gap* (*i.e.,* correlating low-level artifacts into high-level actions) to both raise alerts and rewrite the analyzed protocol inline to mitigate vulnerabilities. An unsupervised learning technique that could have its sensitivity and reactivity tuned based on the application could be a valuable tool for critical embedded systems. This work would create more survivable and resilient embedded systems.

7.2 Unglitchable Hardware

Glitching still remains one of the most effective methods for attackers to compromise embedded systems. While my work raises the bar significantly by both shining light on how these attacks work in practice and providing a framework to easily deploy effective defenses on real-world systems (*i.e.,* GlitchResistor), it does not completely mitigate the problem. In fact, software-based techniques can never completely mitigate glitching attacks. In the limit every defensive instruction could be "skipped." To this end, the ability to glitch a systems appears to be directly related to the hamming distance between the intended instruction and a nearby *no operation* instruction, which the intended instruction can transformed into by a glitch. Thus, it appears that ISAs could be constructed in a glitch resistant way by simply remapping instructions. There are also novel fabrication techniques that can be done to ensure that glitches are always detected (*e.g.,* brown out detection, redundant checks, or parallel computation with different clock domains). Exploring the fabrication of glitch-resistant hardware could produce embedded systems that would be significantly more secure against physical attacks than the systems that we are using today.

7.3 Automated Embedded System Emulation

Emulating, or virtualizing embedded, systems is one of the biggest problems that the security community currently faces. Indeed, Conware makes this possible by providing a framework that can take recordings from real hardware and create robust, automata-based models that can be used to support full-system emulation of embedded systems. However, Conware has multiple limitations. For example, the current models can be too specific in some cases, which can lead to the models returning sub-optimal results in emulation. More precisely, in our experiments, our models for UART would have a chain of edges and states for each buffered write, which would be the exact length that was observed in practice. Thus, if the emulated firmware printed a string of a different length to UART it would either dump the buffer too soon or too late, compared to the actual hardware. Similarly, Conware only merges edges (*i.e.,* state transitions) that will not violate any of the semantics from the recording. Realistically, more states could likely be merged, without negatively impacting the functionality of the model.

Conware was designed in this way to allow us to verify that the general idea (*i.e.,* generating usable automata from recordings) was sound, without undue complexity. There is still a lot of room for improvement on this general technique. In the case of the interrupt-based UART buffer, the problem arises from the fact that interrupts are currently triggered on incoming edges to generalize the approach. However, a technique could be developed to detect the cases where a MMIO write is *always* followed by a specific interrupt, and handle those accordingly. This would completely eliminate the over-specific automata that we saw in our experiment. An even more interesting approach would be to incorporate reinforcement learning and static analysis into the framework to automatically make the model as general as possible, while remaining useful. The idea is to train the model on *similar* hardware initially, and then iteratively merge states in the model to create a more generalized automata, and test it against the target firmware. By incorporating static analysis, *new* states could even be added to the automata in the reinforcement-learning phase by analyzing the binary on the fly to minimize the search space of potential values to return for the future MMIO reads. A combination of these techniques could decrease the dependence on hardware recordings, or even eliminate the requirement entirely, producing a generalized framework for automatically emulating any embedded system.

Chapter 8

Final Thoughts

A creation is only as useful, or as good, as its effect on the world. Computers, like any world-altering technology, have the potential to either be one of the greatest human creations, or the worst, depending on how we use them. Ubiquitous embedded systems are set to change the human experience forever. I hope that my work, both past and future, is able to help create a future where humans no longer fear that their computer systems will be "hacked," but can instead enjoy these incredible creations for what they are: an awesome piece of engineering that can be used to improve the human experience.

Bibliography

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et. al.*, *Understanding the mirai botnet*, in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1093–1110, 2017.

[2] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, *Ether: malware analysis via hardware virtualization extensions*, in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 51–62, ACM, 2008.

[3] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, *The cuckoo sandbox*, *Accessed: Dec* **16** (2012) 2018.

[4] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, *BitBlaze: A New Approach to Computer Security via Binary Analysis*, in *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, Springer, 2008.

[5] Anubis, "Analyzing Unknown Binaries." `http://anubis.iseclab.org`.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, *A few billion lines of code later: using static analysis to find bugs in the real world*, *Communications of the ACM* **53** (2010), no. 2 66–75.

[7] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, *Comparing bug finding tools with reviews and tests*, in *IFIP International Conference on Testing of Communicating Systems*, pp. 40–55, Springer, 2005.

[8] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov, *Configurable toolset for static verification of operating systems kernel modules*, *Programming and Computer Software* **41** (2015), no. 1 49–64.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-aware evolutionary fuzzing.*, in *NDSS*, vol. 17, pp. 1–14, 2017.

[10] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting fuzzing through selective symbolic execution.*, in *NDSS*, vol. 16, pp. 1–16, 2016.

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, *Evaluating fuzz testing*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, ACM, 2018.

[12] P. Wagle, C. Cowan, *et. al.*, *Stackguard: Simple stack smash protection for gcc*, in *Proceedings of the GCC Developers Summit*, pp. 243–255, Citeseer, 2003.

[13] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, *On the effectiveness of address-space randomization*, in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 298–307, ACM, 2004.

[14] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, *Control-flow integrity principles, implementations, and applications*, *ACM Transactions on Information and System Security (TISSEC)* **13** (2009), no. 1 4.

[15] D. Kirat, G. Vigna, and C. Kruegel, *BareBox: Efficient Malware Analysis on Bare-metal*, in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, ACM, 2011.

[16] D. Kirat, G. Vigna, and C. Kruegel, *Barecloud: bare-metal analysis-based evasive malware detection*, in *Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14). USENIX Association, Berkeley, CA, USA*, pp. 287–301, 2014.

[17] K. Koscher, T. Kohno, and D. Molnar, {*SURROGATES*}*: Enabling near-real-time dynamic analyses of embedded systems*, in *9th* {*USENIX*} *Workshop on Offensive Technologies (*{*WOOT*} *15)*, 2015.

[18] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, *What you corrupt is not what you crash: Challenges in fuzzing embedded devices.*, in *NDSS*, 2018.

[19] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, *Avatar 2: A multi-target orchestration platform*, in *Workshop on Binary Analysis Research (colocated with NDSS Symposium)(February 2018), BAR*, vol. 18, 2018.

[20] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, *et. al.*, *Avatar: A framework to support dynamic security analysis of embedded systems' firmwares.*, in *NDSS*, pp. 1–16, 2014.

[21] B. Feng, A. Mera, and L. Lu, *P 2 im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling*, .

[22] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, *Repeatable reverse engineering with panda*, in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, p. 4, ACM, 2015.

[23] D. D. Chen, M. Woo, D. Brumley, and M. Egele, *Towards automated dynamic analysis for linux-based embedded firmware.*, in *NDSS*, pp. 1–16, 2016.

[24] C. Spensky, H. Hu, and K. Leach, *Lo-phi: Low-observable physical host instrumentation for malware analysis.*, in *NDSS*, 2016.

[25] H. Hu and C. S. Spensky, *Systems and methods for single device authentication*, Oct. 26, 2017. US Patent App. 15/178,320.

[26] C. Spensky, J. Stewart, A. Yerukhimovich, R. Shay, A. Trachtenberg, R. Housley, and R. K. Cunningham, *Sok: Privacy on mobile devices–it's complicated*, *Proceedings on Privacy Enhancing Technologies* **2016** (2016), no. 3 96–116.

[27] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, *Boomerang: Exploiting the semantic gap in trusted execution environments.*, in *NDSS*, 2017.

[28] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, *Periscope: An effective probing and fuzzing framework for the hardware-os boundary.*, in *NDSS*, 2019.

[29] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *{DR}.{CHECKER}: A soundy analysis for linux kernel drivers*, in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1007–1024, 2017.

[30] C. Spensky, A. Machiry, M. Busch, K. Leach, R. Housley, C. Kruegel, and G. Vigna, *TRUST.IO: protecting physical interfaces on cyber-physical systems*, in *2020 IEEE Conference on Communications and Network Security (CNS) (IEEE CNS 2020)*, (Avignon, France), June, 2020.

[31] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, *Combined software and hardware attacks on the java card control flow*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 283–296, Springer, 2011.

[32] G. Barbu, H. Thiebeauld, and V. Guerin, *Attacks on java card 3.0 combining fault and logical attacks*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 148–163, Springer, 2010.

[33] G. Barbu, G. Duc, and P. Hoogvorst, *Java card operand stack: fault attacks, combined attacks and countermeasures*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 297–313, Springer, 2011.

[34] F. Project, "The xbox 360 reset glitch hack." `https://free60project.github.io/wiki/Reset_Glitch_Hack.html`.

[35] N. Lawson, "How the ps3 hypervisor was hacked." `https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/`, 2010.

[36] Y. Lu and Davee, "Viva la vita vida: Hacking the most secure handheld console." `https://media.ccc.de/v/35c3-9364-viva_la_vita_vida`, December, 2018.

[37] G. T. H. G. Roussel-Tarbouriech, N. Menard, T. True, *et. al.*, *Methodically defeating nintendo switch security*, *arXiv preprint arXiv:1905.07643* (2019).

[38] A. Galauner, "Glitching the switch." `https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch#t=82`, June, 2018.

[39] A. Cui and R. Housley, *BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection*, in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[40] Y. Lu, "Attacking hardware aes with dfa." `https://yifan.lu/images/2019/02/Attacking_Hardware_AES_with_DFA.pdf`, Februrary, 2019.

[41] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, *Plundervolt: Software-based fault injection attacks against intel sgx*, in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[42] T. roth, *Trustzone-m(eh): Breaking armv8-m's security*, in *The 36th Chaos Communication Congress (36C3)*, December, 2019.

[43] C. Miller and C. Valasek, *Remote exploitation of an unaltered passenger vehicle*, *Black Hat USA* **2015** (2015) 91.

[44] N. Wiersma and R. Pareja, *Safety!= security: On the resilience of asil-d certified microcontrollers against fault injection attacks*, in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 9–16, IEEE, 2017.

[45] A. Abbasi and M. Hashemi, *Ghost in the PLC: Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack*, *BlackHat Europe* (2016).

[46] N. Falliere, L. O. Murchu, and E. Chien, *W32. Stuxnet Dossier*, *White paper, Symantec Corp., Security Response* **5** (2011), no. 6.

[47] S. Gayou, “Remote Code Execution on the Smiths Medical Medfusion 4000.” `https://github.com/sgayou/medfusion-4000-research/blob/master/doc/README.md`, January, 2018.

[48] G. Wassermann, “NXP Semiconductors MQX RTOS Contain Multiple Vulnerabilities.” `https://www.kb.cert.org/vuls/id/590639`, October, 2017.

[49] J. Radcliffe, *Hacking Medical Devices for fun and insulin: Breaking the human SCADA systemun and Insulin: Breaking the Human SCADA System*, in *BlackHat*, vol. 2011, 2011.

[50] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, *Detecting environment-sensitive malware*, in *Recent Advances in Intrusion Detection*, pp. 338–357, Springer, 2011.

[51] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, *Efficient detection of split personalities in malware.*, in *NDSS*, 2010.

[52] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, *Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware*, in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 177–186, IEEE, 2008.

[53] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters, “Volatility framework - volatile memory extraction utility framework.”

[54] B. Carrier, “The Sleuth Kit.” `http://www.sleuthkit.org/sleuthkit/desc.php`, July, 2015.

[55] M. Dornseif, *0wned by an ipod*, *Presentation, PacSec* (2004).

[56] B. D. Carrier and J. Grand, *A hardware-based memory acquisition procedure for digital investigations*, *Digital Investigation* **1** (2004), no. 1 50–60.

[57] J. Wang, F. Zhang, K. Sun, and A. Stavrou, *Firmware-assisted memory acquisition and analysis tools for digital forensics*, in *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pp. 1–5, IEEE, 2011.

[58] J. Mankin and D. Kaeli, *Dione: a flexible disk monitoring and analysis framework*, in *Research in Attacks, Intrusions, and Defenses*, pp. 127–146. Springer, 2012.

[59] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, *Ether: Malware Analysis via Hardware Virtualization Extensions*, in *Proceedings of the 15th Annual Conference on Computer and Communications Security (CCS)*, ACM, 2008.

[60] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, *V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis*, in *Proceedings of the 8th SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, ACM, 2012.

[61] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, *Dynamic and Transparent Analysis of Commodity Production Systems*, in *Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10)*, IEEE/ACM, 2010.

[62] Z. Deng, X. Zhang, and D. Xu, *Spider: stealthy binary program instrumentation and debugging via hardware virtualization*, in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 289–298, ACM, 2013.

[63] T. Garfinkel, *Traps and pitfalls: Practical problems in system call interposition based security tools.*, in *NDSS*, vol. 3, pp. 163–176, 2003.

[64] B. Hay and K. Nance, *Forensics examination of volatile system data using virtual introspection*, *ACM SIGOPS Operating Systems Review* **42** (2008), no. 3 74–82.

[65] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, *Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring*, in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 363–374, ACM, 2011.

[66] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, *Shellos: Enabling fast detection and forensic analysis of code injection attacks.*, in *USENIX Security Symposium*, 2011.

[67] X. Jiang, X. Wang, and D. Xu, *Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction*, in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 128–138, ACM, 2007.

[68] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, *Tappan zee (north) bridge: mining memory accesses for introspection*, in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 839–850, ACM, 2013.

[69] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, *Terra: A virtual machine-based platform for trusted computing*, in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 193–206, ACM, 2003.

[70] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, *Virtual machine introspection in a hybrid honeypot architecture.*, in *CSET*, 2012.

[71] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, *Sok: Introspections on trust and the semantic gap*, in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 605–620, IEEE, 2014.

[72] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, *Dimsum: Discovering semantic data of interest from un-mappable memory with confidence*, in *Proc. ISOC Network and Distributed System Security Symposium*, 2012.

[73] Y. Fu and Z. Lin, *Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection*, in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 586–600, IEEE, 2012.

[74] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, *Virtuoso: Narrowing the semantic gap in virtual machine introspection*, in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 297–312, IEEE, 2011.

[75] S. Krishnan, K. Z. Snow, and F. Monrose, *Trail of bytes: efficient support for forensic analysis*, in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 50–60, ACM, 2010.

[76] T. Raffetseder, C. Kruegel, and E. Kirda, *Detecting System Emulators*, in *Information Security*. Springer, 2007.

[77] J. Rutkowska, "Red Pill." `http://www.ouah.org/Red_Pill.html`.

[78] D. Quist, V. Smith, and O. Computing, *Detecting the presence of virtual machines using the local data table*, *Offensive Computing* (2006).

[79] P. Ferrie, *Attacks on more virtual machine emulators*, *Symantec Technology Exchange* (2007).

[80] J. Wang, A. Stavrou, and A. Ghosh, *Hypercheck: A hardware-assisted integrity monitor*, in *Recent Advances in Intrusion Detection*, pp. 158–177, Springer, 2010.

[81] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, *Hypersentry: enabling stealthy in-context measurement of hypervisor integrity*, in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 38–49, ACM, 2010.

[82] F. Zhang, K. Leach, K. Sun, and A. Stavrou, *Spectre: A dependable introspection framework via system management mode*, in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2013.

[83] A. Baliga, V. Ganapathy, and L. Iftode, *Automatic inference and enforcement of kernel data structure invariants*, in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 77–86, IEEE, 2008.

[84] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, *Copilot-a coprocessor-based kernel runtime integrity monitor.*, in *USENIX Security Symposium*, pp. 179–194, San Diego, USA, 2004.

[85] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, *Secure coprocessor-based intrusion detection*, in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pp. 239–242, ACM, 2002.

[86] J. Molina and W. Arbaugh, *Using independent auditors as intrusion detection systems*, in *Information and Communications Security*, pp. 291–302. Springer, 2002.

[87] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, *Vigilare: toward snoop-based kernel integrity monitor*, in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 28–37, ACM, 2012.

[88] B. Schatz, *Bodysnatcher: Towards reliable volatile memory acquisition by software*, *digital investigation* **4** (2007) 126–134.

[89] M. Cohen, D. Bilby, and G. Caronni, *Distributed forensics and incident response in the enterprise*, *digital investigation* **8** (2011) S101–S110.

[90] A. Martin, *Firewire memory dump of a windows xp computer: a forensic approach*, *Black Hat DC* (2007) 1–13.

[91] J. Stüttgen and M. Cohen, *Anti-forensic resilient memory acquisition*, *Digital Investigation* **10** (2013) S105–S115.

[92] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, *Dksm: Subverting virtual machine introspection for fun and profit*, in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pp. 82–91, IEEE, 2010.

[93] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, *Down to the bare metal: Using processor features for binary analysis*, in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 189–198, ACM, 2012.

[94] J. Rutkowska, *Beyond the cpu: Defeating hardware based ram acquisition*, *Proceedings of BlackHat DC 2007* (2007).

[95] D. Aumaitre and C. Devine, *Subverting windows 7 x64 kernel with dma attacks*, *HITBSecConf 2010 Amsterdam* **29** (2010).

[96] P. Stewin and I. Bystrov, *Understanding dma malware*, in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41. Springer, 2013.

[97] J. Heasman, *Implementing and detecting a pci rootkit*, *Retrieved February* **20** (2006), no. 2007 3.

[98] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain, *Can you still trust your network card*, *CanSecWest/core10* (2010) 24–26.

[99] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, *You can type, but you can't hide: A stealthy gpu-based keylogger*, in *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[100] J. Bowling, *Clonezilla: build, clone, repeat*, *Linux journal* **2011** (2011), no. 201 6.

[101] F. Bellard, *Qemu, a fast and portable dynamic translator.*, in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[102] I. Habib, *Virtualization with kvm*, *Linux Journal* **2008** (2008), no. 166 8.

[103] B. D. Payne, "Libvmi: Simplified virtual machine introspection."

[104] "libvirt: The virtualization api." `http://libvirt.org/`, 06, 2014.

[105] Altera Corporation, "PCI Express High Performance Reference Design." `http://www.altera.com/literature/an/an456.pdf`, 2014.

[106] J. FitzPatrick and M. Crabill, *NSA Playset: PCIE*, in *DEFCON 22*, 2014.

[107] P. Stewin, *A Primitive for Revealing Stealthy Peripheral-based Attacks on the Computing Platform's Main Memory*, in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*, pp. 1–20, Springer, 2013.

[108] "Iozone filesystem benchmark." `http://www.iozone.org/`, 2006.

[109] J. Wang, K. Sun, and A. Stavrou, *A dependability analysis of hardware-assisted polling integrity checking systems*, in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[110] B. Dees, *Native command queuing-advanced performance in desktop storage*, *Potentials, IEEE* **24** (2005), no. 4 4–7.

[111] M. Cohen and J. Metz, "Pytsk." `https://github.com/py4n6/pytsk`, 2014.

[112] B. Blunden, *The Rootkit Arsenal.* Jones and Barlett Learning, 2 ed., 2013.

[113] "An overview of exploit packs." http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html, May, 2015.

[114] A. Ortega, "Paranoid fish." `http://github.com/a0rtega/pafish`.

[115] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, *et. al.*, *Toward the analysis of embedded firmware through automated re-hosting*, in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pp. 135–150, 2019.

[116] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, *Halucinator: Firmware re-hosting through abstraction layer emulation*, .

[117] F. Comert and T. Ovatman, *Attacking state space explosion problem in model checking embedded tv software*, *IEEE Transactions on Consumer Electronics* **61** (2015), no. 4 572–579.

[118] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, *Avatar2: A multi-target orchestration platform*, in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, pp. 1–11, 2018.

[119] B. Feng, A. Mera, and L. Lu, *P^2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling (extended version)*, *arXiv preprint arXiv:1909.06472* (2019).

[120] G. Gracioli and S. Fischmeister, *Tracing and recording interrupts in embedded software*, *Journal of Systems Architecture* **58** (2012), no. 9 372–385.

[121] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, *Simics: A full system simulation platform*, *Computer* **35** (2002), no. 2 50–58.

[122] M. Samek, "State machines for event-driven systems." `https://barrgroup.com/embedded-systems/how-to/state-machines-event-driven-systems`, May, 2016.

[123] J. Hopcroft, *An n log n algorithm for minimizing states in a finite automaton*, in *Theory of machines and computations*, pp. 189–196. Elsevier, 1971.

[124] L. Where Labs, "Bus pirate." `http://dangerousprototypes.com/docs/Bus_Pirate`, October, 2019.

[125] J. Grand and J. Friday, *Advanced hardware hacking techniques*, *DEFCON* **12** (2004) 59.

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

[127] C. Lattner, *Llvm and clang: Next generation compiler technology*, in *The BSD conference*, vol. 5, 2008.

[128] J. Ganssle, *Reentrancy*, in *The Firmware Handbook*, pp. 231–244. Elsevier, 2004.

[129] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Karonte: Detecting insecure multi-binary interactions in embedded firmware*, in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 431–448.

[130] Atmel, "SAM3X/ SAM3A Series (DATASHEET)." `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf`, March, 2015.

[131] A. Limited, "Cortex-m3 technical reference manual (revision r2p1)." `http://users.ece.utexas.edu/~valvano/EE345L/Labs/Fall2011/CortexM3_TRM_r2p1.pdf`, July, 2010.

[132] Geeetech, "Arduino IR Remote Control." `http://www.geeetech.com/wiki/index.php/Arduino_IR_Remote_Control`, May, 2012.

[133] BARRAGAN, "Sweep." `https://www.arduino.cc/en/Tutorial/Sweep`, November, 2013.

[134] Altium, "Nec infrared transmission protocol." `https://techdocs.altium.com/display/FPGA/NEC+Infrared+Transmission+Protocol`, September, 2017.

[135] D. Benson, "Arduino as a rapid prototyping system." `https://www.embedded.com/arduino-as-a-rapid-prototyping-system/`, August, 2015.

[136] J. Beningo, "Prototype to production: Arduino for the professional." `https://www.edn.com/prototype-to-production-arduino-for-the-professional/`, May, 2016.

[137] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et. al.*, *Sok:(state of) the art of war: Offensive techniques in binary analysis*, in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, IEEE, 2016.

[138] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, *Fault injection techniques and tools*, *Computer* **30** (1997), no. 4 75–82.

[139] M. Witteman and M. Oostdijk, *Secure application programming in the presence of side channel attacks*, in *RSA conference*, vol. 2008, 2008.

[140] M. Dadashi, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, *Hardware-software integrated diagnosis for intermittent hardware faults*, in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 363–374, IEEE, 2014.

[141] Y. Lu, *Injecting software vulnerabilities with voltage glitching*, *arXiv preprint arXiv:1903.08102* (2019).

[142] C. Bozzato, R. Focardi, and F. Palmarini, *Shaping the glitch: Optimizing voltage fault injection attacks*, *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019) 199–224.

[143] S. P. Skorobogatov and R. J. Anderson, *Optical fault induction attacks*, in *International workshop on cryptographic hardware and embedded systems*, pp. 2–12, Springer, 2002.

[144] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, *Practical optical fault injection on secure microcontrollers*, in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 91–99, IEEE, 2011.

[145] J. Balasch, B. Gierlichs, and I. Verbauwhede, *An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus*, in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 105–114, IEEE, 2011.

[146] S. Ordas, L. Guillaume-Sage, and P. Maurine, *Electromagnetic fault injection: the curse of flip-flops*, *Journal of Cryptographic Engineering* **7** (2017), no. 3 183–197.

[147] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, *Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller*, in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 77–88, IEEE, 2013.

[148] E. DeBusschere and M. McCambridge, *Modern game console exploitation*, *Technical Report, Department of Computer Science, University of Arizona* (2012).

[149] N. Timmers and A. Spruyt, *Bypassing secure boot using fault injection*, *Black Hat Europe* **2016** (2016).

[150] N. Timmers and C. Mune, *Escalating privileges in linux using voltage fault injection*, in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 1–8, IEEE, 2017.

[151] N. Timmers, A. Spruyt, and M. Witteman, *Controlling pc on arm using fault injection*, in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 25–35, IEEE, 2016.

[152] J. Gratchoff, N. Timmers, A. Spruyt, and L. Chmielewski, *Proving the wild jungle jump*, .

[153] C. O'Flynn and Z. D. Chen, *Chipwhisperer: An open-source platform for hardware embedded security research*, in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pp. 243–260, Springer, 2014.

[154] J. W. Duran and S. Ntafos, *A report on random testing*, in *Proceedings of the 5th international conference on Software engineering*, pp. 179–183, IEEE Press, 1981.

[155] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, *The sorcerer's apprentice guide to fault attacks*, *Proceedings of the IEEE* **94** (2006), no. 2 370–382.

[156] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, *Sok: sanitizing for security*, *IEEE Security and Privacy* (2019).

[157] "American fuzzy lop." `http://lcamtuf.coredump.cx/afl/`.

[158] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, *Spectre attacks: Exploiting speculative execution*, *arXiv preprint arXiv:1801.01203* (2018).

[159] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown*, *arXiv preprint arXiv:1801.01207* (2018).

[160] M. Seaborn and T. Dullien, *Exploiting the dram rowhammer bug to gain kernel privileges*, *Black Hat* **15** (2015).

[161] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, *Flipping bits in memory without accessing them: An experimental study of dram disturbance errors*, in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 361–372, IEEE Press, 2014.

[162] R. Omarouayache, J. Raoult, S. Jarrix, L. Chusseau, and P. Maurine, *Magnetic microprobe design for em fault attack*, in *2013 International Symposium on Electromagnetic Compatibility*, pp. 949–954, IEEE, 2013.

[163] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, *Low voltage fault attacks on the rsa cryptosystem*, in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 23–31, IEEE, 2009.

[164] J.-M. Schmidt and C. Herbst, *A practical fault attack on square and multiply*, in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 53–58, IEEE, 2008.

[165] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, *Low voltage fault attacks to aes*, in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 7–12, IEEE, 2010.

[166] C. O'Flynn, *Fault injection using crowbars on embedded systems.*, *IACR Cryptology ePrint Archive* **2016** (2016) 810.

[167] A. G. Yanci, S. Pickles, and T. Arslan, *Detecting voltage glitch attacks on secure devices*, in *2008 Bio-inspired, Learning and Intelligent Systems for Security*, pp. 75–80, IEEE, 2008.

[168] F. Rodríguez, J. C. Campelo, and J. J. Serrano, *A watchdog processor architecture with minimal performance overhead*, in *International Conference on Computer Safety, Reliability, and Security*, pp. 261–272, Springer, 2002.

[169] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, *Hardware-assisted run-time monitoring for secure program execution on embedded processors*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **14** (2006), no. 12 1295–1308.

[170] F. Rodríguez and J. J. Serrano, *Control flow error checking with isis*, in *International Conference on Embedded Software and Systems*, pp. 659–670, Springer, 2005.

[171] M. Werner, E. Wenger, and S. Mangard, *Protecting the control flow of embedded processors against fault attacks*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 161–176, Springer, 2015.

[172] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, *Experimental evaluation of two software countermeasures against fault attacks*, in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 112–117, IEEE, 2014.

[173] A. Spruyt, *Building fault models for microcontrollers*, *University of Amsterdam, Amsterdam, Tech. Rep* (2012) 2011–2012.

[174] G. Thessalonikefs, *Electromagnetic fault injection characterization*, .

[175] T. Korak and M. Hoefler, *On the effects of clock and power supply tampering on two microcontroller platforms*, in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 8–17, IEEE, 2014.

[176] L. Zussa, J.-M. Dutertre, J. Clediere, and A. Tria, *Power supply glitch induced faults on fpga: An in-depth analysis of the injection mechanism*, in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pp. 110–115, IEEE, 2013.

[177] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, *High precision fault injections on the instruction cache of armv7-m architectures*, in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 62–67, IEEE, 2015.

[178] M. Hutter and J.-M. Schmidt, *The temperature side channel and heating fault attacks*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 219–235, Springer, 2013.

[179] T. Korak, M. Hutter, B. Ege, and L. Batina, *Clock glitch attacks in the presence of heating*, in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 104–114, IEEE, 2014.

[180] A. Q. Nguyen and H. V. Dang, *Unicorn: Next generation cpu emulator framework*, in *Proceedings of the 2015 Blackhat USA conference*, 2015.

[181] N. A. Quynh, *Capstone: Next generation disassembly framework*, *Black Hat USA* (2014).

[182] N. A. Quynh, *Keystone: Next generation assembler framework*, *Black Hat USA* (2016).

[183] A. ARM7TDMI, *Technical reference manual*, *Advanced RISC Machines Ltd.,(15 May 2003).*

[184] C.-K. Chang, G. Li, and M. Erez, *Evaluating compiler ir-level selective instruction duplication with realistic hardware errors*, in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pp. 41–49, IEEE, 2019.

[185] M. Milshtein, *A new two-error-correcting binary code of length 16*, *Cryptography and Communications* (2019) 1–5.

[186] M. Lubinets, "Reed solomon bch encoder and decoder." `https://github.com/mersinvald/Reed-Solomon`, Februrary, 2016.

[187] L. Cojocar, K. Papagiannopoulos, and N. Timmers, *Instruction duplication: Leaky and not too fault-tolerant!*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 160–179, Springer, 2017.

[188] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, *Modeling soft-error propagation in programs*, in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 27–38, IEEE, 2018.

[189] G. Li, Q. Lu, and K. Pattabiraman, *Fine-grained characterization of faults causing long latency crashes in programs*, in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 450–461, IEEE, 2015.

[190] T. Chen, "Guarding against physical attacks: The xbox one story." `https://www.platformsecuritysummit.com/2019/speaker/chen/`, October, 2019.

[191] A. Vasudevan and R. Yerraballi, *Stealth Breakpoints*, in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[192] N. A. Quynh and K. Suzaki, *Virt-ICE: Next-generation Debugger for Malware Analysis*, in *Black Hat USA*, 2010.

[193] IDA Pro. `www.hex-rays.com/products/ida/`.

[194] "OllyDbg." `www.ollydbg.de`.

[195] D. Bruening, Q. Zhao, and S. Amarasinghe, *Transparent Dynamic Instrumentation*, in *Proceedings of the 8th Conference on Virtual Execution Environments (VEE)*, ACM SIGPLAN/SIGOPS, 2012.

[196] "Windbg." `www.windbg.org`.

[197] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, *Charm: Facilitating dynamic analysis of device drivers of mobile systems*, in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 291–307, 2018.

[198] M. Kammerstetter, C. Platzer, and W. Kastner, *Prospect: peripheral proxying supported embedded code testing*, in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 329–340, ACM, 2014.

[199] M. Kammerstetter, D. Burian, and W. Kastner, *Embedded security testing with peripheral device caching and runtime program state approximation*, in *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[200] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, M. Grace, R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar, *et. al.*, *Partemu: Enabling dynamic analysis of real-world trustzone software using emulation*, in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)(To Appear)*, 2020.

[201] I. Pustogarov, Q. Wu, and D. Lie, *Ex-vivo dynamic analysis framework for android device drivers*, .

[202] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, *Formal verification of a software countermeasure against instruction skip attacks*, *Journal of Cryptographic Engineering* **4** (2014), no. 3 145–156.

[203] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, *Countermeasures against fault attacks on software implemented aes: effectiveness and cost*, in *Proceedings of the 5th Workshop on Embedded Systems Security*, p. 7, ACM, 2010.

[204] J.-F. Lalande, K. Heydemann, and P. Berthomé, *Software countermeasures for control flow integrity of smart card c codes*, in *European Symposium on Research in Computer Security*, pp. 200–218, Springer, 2014.

[205] Z. Chen, J. Shen, A. Nicolau, A. Veidenbaum, N. F. Ghalaty, and R. Cammarota, *Camfas: A compiler approach to mitigate fault attacks via enhanced simdization*, in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 57–64, IEEE, 2017.

[206] R. J. Gove, K. Balmer, N. K. Ing-Simmons, and K. M. Guttag, *Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation*, May 18, 1993. US Patent 5,212,777.

[207] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, *Obfuscator-LLVM – software protection for the masses*, in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015* (B. Wyseur, ed.), pp. 3–9, IEEE, 2015.

[208] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, *A virtual fault injection framework for reliability-aware software development*, in *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, pp. 69–74, IEEE, 2015.

[209] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza, *Soft error injection methodology based on qemu software platform*, in *2014 15th Latin American Test Workshop-LATW*, pp. 1–5, IEEE, 2014.

[210] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, *Comprehensive analysis of software countermeasures against fault attacks*, in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 404–409, IEEE, 2013.