University of California
Santa Barbara

Discovery and Remediation of Vulnerabilities in Monolithic IoT Firmware

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

Doctor of Philosophy
in
Computer Science

by

Eric D. Gustafson

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Chandra Krintz

September 2020

The Dissertation of Eric D. Gustafson is approved.

Professor Chandra Krintz

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

June 2020

Discovery and Remediation of Vulnerabilities in Monolithic IoT Firmware

by

Eric D. Gustafson

Acknowledgements

The content of Chapter 3 is based upon work supported by the National Science Foundation under Award No. CNS-1704253, and by the Office of Naval Research under Award # N00014-17-1-2011. Additionally, this work was in part funded by a research contract with Siemens AG.

The content of Chapter 4 is based upon work supported by ONR under Award No. N00014-17-1-2011 and N00014-17-1-2513, by NSF under award numbers CNS-1718637, CNS-1704253 and CNS-1801601, by AFRL and DARPA under agreement number FA8750-19-C-0003, by the US Department of Homeland Security under agreement number FA8750-19-2-0005, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850868).

Portions of this work were supported by Sandia National Laboratories, a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Department of Energy, Sandia National Laboratories, Department of Homeland Security, Office of Naval Research, AFRL, DARPA, the U.S. Government, or the European Research Council.

Curriculum Vitæ

Eric D. Gustafson

Education

2020	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2014	M.S. in Computer Science, University of California, Davis.
2011	B.S. in Computer Science, California Polytechnic State University, San Luis Obispo.

Publications

1. **Abraham Clements and Eric Gustafson**, Tobias Scharnowski, Paul Grossen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, Mathias Payer. *HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation.* Proceedings of the 29th USENIX Security Symposium (USENIX '20).
2. **Eric Gustafson**, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Aurelien Francillon, Davide Balzarotti, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna. *Toward the Analysis of Embedded Firmware through Automated Re-hosting.* Proceedings of the 22nd International Symposeum on Research in Attacks, Intrusions, and Defenses (RAID '19).
3. Shirin Nilizadeh, Hojjat Aghakhani, **Eric Gustafson**, Christopher Kruegel, and Giovanni Vigna. *Think Outside the Dataset: Finding Fraudulent Reviews using Cross-Dataset Analysis.* Proceedings of the 2019 World Wide Web Conference (WWW '19)
4. Machiry, Aravind, Nilo Redini, **Eric Gustafson**, Hojjat Aghakhani, Christopher Kruegel, and Giovanni Vigna. *Towards Automatically Generating a Sound and Complete Dataset for Evaluating Static Analysis Tools.* Proceedings of the NDSS Workshop of Binary Analysis Research (BAR '19).
5. Machiry Aravind, Nilo Redini, **Eric Gustafson**, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. *Using Loops For Malware Classification Resilient to Feature-unaware Perturbations.* Proceedings of the Annual Computer Security Applications Conference (ACSAC '18).
6. Antonio Bianchi, **Eric Gustafson**, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna. *Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information.* Proceedings of the Annual Computer Security Applications Conference (ACSAC '2017).
7. Erik Trickel, Francesco Disperati, **Eric Gustafson**, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, Giovanni Vigna. *Shell We Play A Game? CTF-as-a-service for Security Education.* Proceedings of the 2017 USENIX Workshop on Advances in Security Education (ASE '17)

8. Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, **Eric Gustafson**, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. *BootStomp: On the Security of Bootloaders in Mobile Devices*. Proceedings of the 26th USENIX Security Symposium (USENIX Security '17).
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*. Proceedings of the Network and Distributed System Security Symposium (NDSS '17).

Abstract

Discovery and Remediation of Vulnerabilities in Monolithic IoT Firmware

by

Eric D. Gustafson

Many recent advances in the scale, cost, and connectivity of hardware have brought about the era of the Internet of Things (IoT), in which numerous objects in our everyday lives now contain networked computing capabilities. Most notably, this has brought with it an astonishing array of new device market sectors, form factors, and use cases, which purport to make our lives easier, simpler, and safer. However, these new network-connected devices have opened a massive new attack surface for attackers to exploit, and a challenging landscape for defenders, which could undermine these benefits.

Unfortunately for analysts, the code of these new ubiquitous, low-power embedded systems increasingly utilize *monolithic firmware images*, in which code, libraries, and data are intermixed, without a conventional operating system or metadata needed by third-party analyses. This combines with the extreme hardware-software coupling found in firmware to create a complex software environment, that is extremely difficult to model for the purposes of conventional program analyses. This has created two significant gaps in the vulnerability discovery lifecycle: modeling of the execution environment, and patching of vulnerabilities, even without a manufacturer's help. As a result, devices with monolithic firmware have been largely ignored by academia and industry thus far.

In this dissertation, we will showcase novel techniques to help bridge the gaps in analysis capabilities between traditional programs and the monolithic firmware of deeply-embedded systems. To overcome the environment modeling problem, we will focus on *re-hosting*, the act of transferring a program from one execution environment to another,

typically from a hardware environment to an emulated one. Re-hosting an important pre-requisite to fuzzing or symbolic execution used for vulnerability discovery, as it allows execution environments to be freely copied and scaled. we will propose two techniques: an automated approach based on observing and modeling the original device's hardware, and a semi-automatic approach based on abstracting away and modeling parts of the firmware itself. we will show how these techniques can allow us to re-host many firmware images, and can be directly used for security analyses to find both synthetic and previously-undiscovered real-world vulnerabilities.

Finally, we will address the issue of patching monolithic firmware. While numerous steps are required for an analyst to produce a final patched firmware image, we focus on automating three critical steps: finding sources of attacker-controlled input, finding a safe location to insert a payload, and locating self-checks intended to thwart modification. We combined these techniques into a system able to produce modified firmware, and used it to correct serious safety-critical issues in three products from the medical, industrial automation, and engineering sectors.

Through both re-hosting and patching, this work completes the vulnerability lifecycle for embedded devices, and helps make our connected world safer.

Contents

Chapter 1

Introduction

Over the last twenty years, advances in wireless networking technology and in the design of embedded systems have led to a shift in the way technology integrates with our daily lives. This movement, known as the Internet of Things (IoT), represents the elimination of the barrier between networked, interactive devices, and more mundane objects, tools, and appliances.

In the last five years, this phenomenon has become tangible to consumers, with the mass-market availability of connected devices for homes and businesses, including thermostats, lighting, physical security, and a variety of sensors. The promise of these products is to enable unprecedented convenience and utility, without the usual complexity and cost of sensing and automation. This increased functionality has been a hit with consumers, with an estimated 5.5 million IoT devices being connected every day in 2016 [1], and over 20 billion in use in 2020.

While these numbers may seem rather high, IoT does not merely describe one class of devices with the same functional or security needs, nor can the user typically achieve any benefit with only one component alone. These devices and services compose an *ecosystem*, combining the means to collect data, make decisions based on this data, and act on it, potentially in a physical manner. One device alone may provide the functionality to close a water valve through an app, and another may be able to detect when water

passes a sensor and notify the user, but only through the combination of those two with communications and logic infrastructure can the user mitigate flood damage in their home. Because the value of an IoT ecosystem is dependent on its ability to incorporate a wide array of devices, the value of a single device is also determined by which, and how many, devices and services it will integrate with. This puts the unusual burden on manufacturers to inter-operate, even without the presence of dominant IoT standards for protocols and design.

This emphasis on heterogeneity and high interconnectedness of devices has resulted in additional implications for the security and privacy of users. Instead of converging on standards, the market has continued to support a collection of protocols, frameworks, and Application Program Interfaces (APIs) as diverse as the devices themselves. Each of these brings with it its own security models, assumptions, and trade-offs, as well as a broadened networked attack surface. This close relationship with the physical world also brings deployment complications (e.g., physical location, intended use), which violate various traditional security assumptions (e.g., physical attacks, observability).

This enticing set of targets has not gone unnoticed by attackers. Security vulnerabilities located in relatively small set of devices allowed for the creation of botnets responsible for some of the largest Distributed Denial of Service (DDoS) attacks to date, targeting journalist Brian Krebs [2], DNS infrastructure used by many websites [3], and the country of Liberia [4]. This botnet leverages weak authentication and poor deployment of certain IoT devices (i.e., Internet-connected cameras) to compromise them, and direct traffic toward a target. The Common Vulnerabilities and Exposures (CVE) group, which catalogs security vulnerabilities, has blamed the rise in IoT-related vulnerabilities for a massive backlog in their workflow [5]. The European Union is even working on a law that would require IoT devices to be accompanied by a security rating [6].

Looked at from a different angle, the result of this IoT trend is that newly-commercialized

embedded systems are now in the hands of consumers, and being used in an increasing number of security and safety-critical applications. Unfortunately, in stark contrast to the desktop and mobile ecosystems, market forces have not created any *de facto* standard for components, protocols, or software, hampering existing program analysis approaches, and making the understanding of each new device an independent, mostly manual, time-consuming effort. At the software level, each new device comes with its unique firmware, which is purpose-built for its specific function, and may not include a conventional operating system. At the hardware level, each device includes its own unique selection of hardware, both on the board (sensors, actuators, etc.) and on the chip (bus controllers, timers, and other I/O peripherals), which combine to form the unique execution environment of the firmware.

Even with rapidly-improving software environments, developers create and test firmware almost entirely on physical testbeds, typically consisting of development versions of the target devices. However, modern software-engineering practices that benefit from scale, such as test-driven development, continuous integration, or fuzzing, are challenging or impractical due to this hardware dependency. In addition, embedded hardware provides limited introspection capabilities, including extremely limited numbers of breakpoints and watchpoints, significantly restricting the ability to perform dynamic analysis on firmware.

The situation for third-party auditors and analysts is even more complex. Manufacturing best-practices dictate stripping out or disabling debugging ports (*e.g.,* JTAG) [7, 8], meaning that many off-the-shelf devices remain entirely opaque. Even if the firmware can be obtained through other means, dynamic analysis remains challenging due to the complex environmental dependencies of the code.

One factor that massively complicates this hardware-software dependency is the format of the firmware itself. Many devices use *monolithic binary firmware*, also know as

"blobs", which pack application code, library code, and data into one binary image, with no discernible metadata or formatting. Therefore, it is difficult for the analyst to distinguish code and data separation, function boundaries, library vs. application code, and the memory map of the execution environment, which are typical prerequisites for common program analyses.

These numerous complexities have caused significant changes in the usual lifecycle of discovering and remediating vulnerabilities found in traditional desktop and mobile software. All aspects of obtaining, analyzing, and eventually patching code have become more difficult as a result of the way hardware and firmware interact. However, the lack of standardization of execution environments (in the case of monolithic firmware, just the device's hardware) has necessitated an additional step: creating a model of this environment to enable tractable, accurate, analyses.

Firmware *re-hosting*–the process of enabling code to run in a different environment from which it was designed–is the primary means by which this gap can be overcome. By re-hosting the firmware into a fully-virtualized, emulated, environment, we can execute firmware at scale through the use of commodity computers, and provide more insight into the execution than is possible on a physical device [9]. Yet, heterogeneity in embedded hardware poses a significant barrier to the useful re-hosting of firmware. The rise of intellectual-property-based, highly-integrated chip designs (*e.g.,* ARM based Systems on Chip – SoC) has resulted in an explosion of available embedded CPUs, whose various on-chip peripherals and memory layouts must be supported in a specialized manner by emulators. However, the popular open-source QEMU emulator supports fewer than 30 ARM devices. Intel's SIMICS [10] supports many CPUs and peripherals, but requires the analyst to manually construct a full model of the system at the hardware level. Worse yet, most embedded systems have other components on their circuit boards that must exist for the firmware to operate, such as sensors, storage devices, or networking components.

Emulation support for these peripherals is virtually nonexistent. Therefore, it is nearly impossible to take an embedded firmware sample and emulate it without significant engineering effort.

Current solutions allowing for the re-hosting of diverse hardware rely on a real specimen of the device, where the emulator forwards interactions with unsupported peripherals to the hardware [11, 12, 13]. Such a "hardware-in-the-loop" approach limits the ability to scale testing to the availability of the original hardware, and offers restricted instrumentation and analysis possibilities compared to what is possible in software. Other techniques [14, 15] focus on recording and subsequently replaying data from hardware, which allows these executions to be scaled and shared, but necessarily requires trace recording from within the device itself, limiting faithful execution in the emulator to just the recorded paths in the program.

Even when bugs are found, and reported, that does not guarantee that a fix will be available. A worrying trend in IoT devices is that devices are being abandoned by vendors [16, 17], or otherwise excluded from support by the vendor [18], and do not receive patches for security issues when they are found. Since embedded systems based on monolithic firmware cannot be simply updated by updating an operating system or libraries, vulnerabilities in these systems may have a much longer patch latency, due to the extra work involved in creating and verifying them. The recent Urgent/11 [19] vulnerabilities affect such a wide variety of real-time operating systems and libraries [20], that it is unclear exactly what the patching situation will be for many of these systems. Unfortunately, the increasingly safety and security critical nature of many of these devices means that users may have no choice but to patch or replace the device. In some cases, replacing the device may not be physically possible or financially practical, and users must take matters into their own hands.

Current solutions tackling binary rewriting work primarily on ELF files [21, 22, 23]. They leverage the metadata found in the files to re-arrange code, fixing code offsets and pointers. Of course, with blob firmware, this metadata is not present, and we are left with only Detours-style patching [24], in which code is inserted into an otherwise-unused space, and the instructions of the program are altered to use it. Worse yet, without metadata, and without an operating system, there is no standard source of input data to the program for a patch to make security-related decisions with. Finding the place to put code in the first place is also difficult, with no guarantees of space to insert code into, minimal on-board storage, and no easy way to determine existing content that's expendable. Finally, in order to make these systems robust, firmware will typically check their content to ensure that it is not intentionally or accidentally modified; this must be overcome before any kind of binary patching can take place. In summary, no technique exists which allows a third-party user or analyst to create patches for a monolithic firmware image without intensive, manual, firmware-specific reverse-engineering.

In this work, we will address the unique challenges preventing the security analysis and remediation of monolithic firmware images, through the use of novel re-hosting and automated program analysis techniques.

We propose to tackle the gap in environment modeling capabilities through the use of novel re-hosting techniques, allowing the virtualization of the firmware with a minimum of analyst effort. In order to help frame these approaches, we identified four aspects of an ideal re-hosting solution, which could allow it to tackle today's diverse embedded systems: A re-hosting scheme must be *virtual* to allow for scale and reduce costs; should also be *interactive*, to allow the firmware to process new input and actually withstand program analysis; should be *abstraction-less* (i.e., it should not rely on high-level concepts, such as operating systems and hardware abstraction layers) to allow the system to handle the

widest possible variety of firmware. Finally, re-hosting should be *automated*, so that the system can overcome the extreme diversity that is impractical for humans to handle. Although previous approaches to the problem are numerous, all are missing at least one of these aspects.

Automated Re-Hosting. The first approach we will present is the first to strive for this kind of idealized automated re-hosting. The technique works by observing the behavior of the original device, then modeling this behavior, creating a set of models which can be plugged into an emulator. We use the system to emulate three hardware devices, with six firmware images, and were able to successfully emulate and automatically test the firmware, and discover synthetic vulnerabilities.

While this approach has the advantage of being mostly hands-off for the analyst, there are a few challenges remaining to be solved when it comes to tackling real-world commercial devices. Aspects of these systems, such as high-frequency interrupts, Direct Memory Access (DMA), and highly-complex peripherals will confound this and similar approaches.

These same challenges complicate both the development and testing of firmware by device manufacturers as well. To mitigate some of these issues, and make their platforms more attractive under tight time-to-market constraints, chip vendors and various third parties provide Hardware Abstraction Layers (HALs). HALs are software libraries that provide high-level hardware operations to the programmer, while hiding details of the particular chip or system on which the firmware executes. This makes porting code between the many similar models from a given vendor, or even between chip vendors, much simpler. Application code written with HALs is therefore, by design, less tightly coupled to the hardware, even when distributed as a monolithic firmware image.

Re-hosting with High Level Emulation. To leverage this observation, the second

re-hosting approach applies re-hosting based on High Level Emulation (HLE), in which these hardware-facing functions are replaced with analyst-created, high-level equivalents. When working with a monolithic firmware image, the first step is to locate the libraries within the firmware using binary analysis. With the function names identified, high level *handlers* (e.g., written a scripting language such as Python) can be quickly created, and re-used for any firmware sample using the same libraries. While this process is neither fully automatic nor abstraction-less, we will show that this trade-off allows our prototype system, HALucinator, to re-host dramatically more complex firmware than other techniques, and keeping the burden on the analyst low. Furthermore, we will show how this technique can be used to fuzz realistic firmware samples, and discover previously-unknown vulnerabilities in firmware and libraries.

Security Retrofitting for Monolithic Firmware. In this final chapter, we explore the challenges of retrofitting monolithic firmware images with new security measures. First, we outline the steps any analyst must take to retrofit firmware, and show that previous work is missing crucial aspects of the process, which are required for a practical solution. We then automate three of these aspects—locating attacker-controlled input, a safe retrofit injection location, and self-checks preventing modification—through the use of novel automated program analysis techniques. We assemble these analyses into a system, Shimware, that is able to guide the analyst in creating a working retrofitted firmware image with a minimal amount of effort and prior knowledge of the device.

To evaluate our system, we employ both a synthetic evaluation and actual retrofitting of three case study devices: a networked bench power supply, a Bluetooth-enabled cardiac implant monitor, and a high-end programmable logic controller (PLC). Not only was our system able to identify the correct sources of input, injection locations, and self-checks, but it injected payloads to correct serious safety and security-critical vulnerabilities in these devices.

1.1 Permissions and Attributions

1. The content of chapter 2 is the result of a collaboration with Marius Muench, Chad Spensky, Nilo Redini, Machiry Aravind, Yanick Fratantonio, Aurelien Francillon, Davide Balzarotti, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2019 edition of the Symposium on Research in Attacks, Instrusions and Defenses (RAID 2019).

2. The content of chapter 3 is the result of a collaboration with Abraham Clements, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer, and will appear in the 2020 edition of the USENIX Security Symposium.

3. The content of chapter 4 is the result of a collaboration with Paul Grosen, Ruoyu Wang, Nilo Redini, Saagar Jha, Sara Rampazzi, Kevin Fu, Christopher Kruegel, and Giovanni Vigna.

Chapter 2

Background and Related Work

2.1 The Vulnerability Lifecycle

In this section, we will frame the concepts presented in this work in the context of how vulnerabilities in software are found and fixed by analysts today. In particular we will discuss how this process differs on embedded systems, and the gaps in our analysis capabilities this creates.

Vulnerability discovery and detection efforts have become a major part of the security efforts of both device and software vendors, and third-party analysts alike. Their combined efforts can help reduce the number of severe bugs available to hostile adversaries, assuming the discovered issues are ethically reported to, and fixed by, the vendors. For third-party analysts, however, discovering new vulnerabilities is also now a lucrative business, with numerous bug bounty programs, contests, and other entities willing to provide significant financial compensation for their efforts.

Regardless of their identities or motives, the overall process used to find and fix vulnerabilities in programs can be broken down into a few key steps: obtaining the code-under-test, performing an analysis to locate potential bugs, confirming these bugs, and creating a patch. In the case of an embedded system, particularly one running a monolithic firmware image, the steps are similar, but with a dramatically higher level of

complexity and difficulty:

1. **Obtaining Code.** The analyst needs to first obtain the code-under-test. Depending on the scenario, this could be source code, or a compiled binary. For common desktop or mobile software, this step is trivial; the program to be analyzed can be simply downloaded or installed via the Internet, an app store, or some other distribution mechanism. However, the firmware of an embedded device was developed for that device only; the manufacturer may only provide the firmware through a specialized update mechanism (e.g., on the device itself, or through a companion app or software package), or in some cases, not at all. Therefore, third-party analysts have the additional tedious step of obtaining this firmware through potentially invasive means, such as attacking the device's storage chips, or interfering with the firmware update mechanism.

2. **Environment Modeling.** In order to perform a meaningful analysis, the analyst must understand or recreate the environment in which the code-under-test is run. When analyzing mobile or desktop applications, this step is generally inherent in the previous step of obtaining the code. An application designed for Android, for example, runs within Android's well-specified environment. Tools are built with the assumptions of this environment in mind, and allow any app built on top of this OS's abstraction to be quickly analyzed in a similar fashion. This is true of all program analyses; dynamic analyses need to be able to execute the program, and static analyses need to know enough about the environment to make meaningful inferences.

   With embedded systems, this step is profoundly more difficult, particularly for devices based on monolithic firmware. The environment that needs to be modeled instead consists of the device's hardware, both within the CPU executing the code,

and elsewhere. We explore this issue in depth in Section 2.2.

3. **Analysis.** With the environment modeled, the analyst can now examine the program for bugs. For third-party analysts, this will typically involve *binary program analyses*, such as fuzzing, static or dynamic taint tracking, and dynamic symbolic execution-based approaches. While these same kinds of analyses are used on embedded systems, they contain numerous new caveats. Firmware does not follow a typical pattern of taking input from or interacting with the user. Many analyses will also need to reason about interrupt events and timing, features typically handled by the operating system in normal programs.

4. **Patching.** With newly-discovered bugs in hand, the focus now turns to preventing them from being abused in the future. This typically involves the vendor re-compiling the software to create a patched version. Of course, this is similar for firmware, but the primary difference is that the device vendor completely controls the update process. Even if the vulnerability is in a library component not written by the vendor, their complicity is typically required to create updated firmware versions.

In summary, when searching for vulnerabilities in a device with a monolithic firmware, there are two major gaps in the typical lifecycle that prevent analysts from readily exploring them. Complexities in modeling the execution environment prevent execution of the code without either the original device or significant manual effort. Patching currently requires the vendor's help, given that source code and correct build environments are typically unavailable. As a result, analysts today often do not analyze monolithic firmware images, due to the immense challenges involved.

In the remainder of this chapter, we will discuss these two issues in more depth. The

following chapters will present novel approaches to both re-hosting and patching to help close these gaps.

2.2 The Re-hosting Problem

To deal with the plethora of software applications that need to be analyzed on desktop and mobile platforms, the security community has developed many techniques for enabling the scalable analysis of programs to find bugs and detect malice. In this section, we examine what makes embedded systems different and much less tractable to these techniques, as well as propose qualities that a system capable of analyzing arbitrary firmware must have.

Today's state-of-the-art program analysis techniques, including dynamic analysis tools such as AFL [25] or symbolic execution engines such as angr [26] or S2E [27], rely on some form of abstraction to be tractable. Dynamic approaches typically rely on virtualization to enable parallel, scalable analyses, while symbolic approaches rely on function summarization of the underlying operating system to minimize the code that they need to execute. In order to use any of these tools, the analyst must take the program out of its original execution environment, and provide a suitable analysis environment able to execute it. This is a process referred to as *re-hosting*.

For desktop and mobile programs, the standardization of the execution environments (e.g., commodity hardware, which consists of a relatively small number of OSes and architectures) has made this re-hosting process simpler. However, with embedded firmware, many well-established assumptions fail. For example, there may not be a general-purpose operating system designed to run arbitrary code on the device, leaving the analyst to deal with the hardware directly. This is especially true for low-power Internet of Things (IoT) devices, which are typically based on microcontroller-class CPUs that lack the ability to

run such OSes. Firmware for these devices is typically obtained in the form of a *binary blob*, an opaque code object containing no metadata about its contents. How this blob is handled is entirely dependent on the CPU hardware, and will vary widely from chip to chip. This also makes distinguishing between library code and device-specific code challenging. With no visible abstractions to use, the execution environment for embedded firmware is the hardware itself. We can break this hardware down into three distinct categories:

- **CPU Core.** The CPU core itself must, of course, be emulated. This includes the instruction set, but also any function able to directly alter code execution, such as the chip's primary interrupt controller.

- **On-Chip Peripherals.** These peripherals include timers, bus controllers, serial ports, General Purpose Input and Output (GPIO), and other features typically included on the die of the CPU itself. Most CPUs expose these peripherals to the program as Memory-Mapped Input/Output (MMIO), where they are organized as a group of contiguous memory locations, that do not behave like normal memory. Each group may contain multiple locations, used for configuring, checking the status of, and exchanging data with the peripheral. An example of a typical MMIO peripheral mapping is shown in Figure 2.1. On-chip peripherals are also responsible for issuing *interrupts*, events that trigger asynchronous changes in control flow in response to a hardware event. More precisely, a peripheral is associated with one or more numbered interrupt "channels" or "lines"; when an interrupt occurs, the code in the firmware associated with that interrupt (known as an Interrupt Service Routine, or ISR) is executed. When, how, and why a peripheral issue interrupts are all properties of the peripheral's hardware on a particular chip, but typically includes the arrival of data, the expiration of timers, and error conditions.

Table 2.1: Excerpt of tools tackling the re-hosting problem

Tool	Virtual	Interactive	Abstraction-less	Automatic
Simics [10]	✓	✓	✓	-
FIE [28]	✓	✓	✓	-
Avatar [29]	-	✓	✓	-
PROSPECT [30, 31]	-	✓	-	✓
Surrogates [32]	-	✓	✓	-
Firmadyne [33]	✓	✓	-	✓
Avatar^2 [34]	✓	✓	✓	-
P2IM [35]	✓	✓	✓	✓
Pretender	✓	✓	✓	✓
HALucinator	✓	✓	✓	-

- **External Peripherals.** These peripherals are the sensors, actuators, and other circuitry on the device's circuit board(s). They are exposed to the program only through one of the on-chip peripherals, including GPIO, or a bus such as Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI). While from the programmer's perspective, communicating with these peripherals is as easy as sending and receiving messages thanks to software libraries, the resulting compiled firmware does so through a complex series of accesses to the MMIO regions of on-chip peripherals, making the direct flow of data in and out of each peripheral difficult to observe. This is also the source of the most variety in embedded systems, as these devices typically contain entirely-custom circuit boards, with whatever array of components the designers felt were necessary.

2.2.1 Aspects of Ideal Re-Hosting

Many solutions have been proposed to enable firmware re-hosting, each with their own qualities and drawbacks. To showcase their differences, we identify four salient properties that an ideal analysis system, capable of handling arbitrary firmware, should

possess. Table 2.1 shows prevalent tools that tackled the re-hosting problem in the past, and classifies them according to the aspects, which are described as follows.

Abstraction-less. An ideal re-hosting solution should not rely on a software abstraction that greatly limits the kinds of firmware on which it can be used. Recently, advances have been made in re-hosting firmware based on the abstractions provided by the Linux OS [33, 36]. Using such an abstraction, when it exists, is advantageous, but it naturally limits the scope of firmware to those that do not have a significant coupling between their primary function and the underlying hardware. Relying on an OS precludes the analysis of, for example, the *blob* firmware we explore in this work. However, as we explore in Chapter 4, even blob firmware samples are often built with an abstraction in mind, although it must be located in the binary in order to be used as such. We therefore show how we can trade the goal of idealized rehosting, which should indeed not rely on an abstraction, for the ability to rehost more complex firmware.

Firmware relying on an abstraction can also be *simulated* without full emulation, if source code is available. Simulators for Contiki [37], mBed [38] and RIOT-OS [39] allow the developer to compile their firmware code into a binary that can run on the host system. In contrast, HALucinator allows for a similar kind of re-hosting to be performed, but on the final firmware binary, and without the availability of source code.

HALucinator draws some inspiration from the work done in game console emulation [40, 41], which pioneered the idea of High Level Emulation, albeit applied to specific hardware environments and software stacks. HALucinator represents a generalization of this idea, and presents the first known application to embedded firmware for security.

Virtual. A re-hosting solution should not depend on the presence of hardware during analysis. Many proposed approaches to firmware analysis [29, 32, 30, 42] require *hardware-in-the-loop* execution. However, such approaches inherently limit the scale of

the analyses. In a dynamic context, only one thread of execution is possible per-device, and re-starting execution, which happens very often in modern fuzzers, can incur a significant time penalty [43]. One way to allow those optimizations, however, is presented by Kammerstetter, et al. [31], who proposed a means of caching requests and responses between a program and the high-level device objects exposed by its Linux-based firmware. While this approach also concerns approximating state, and has a similar goal, it is distinct from the State Approximation approach we present here, as it attempts to understand the state of the program as a proxy for the state of the peripheral, and not the state of the peripheral directly.

Another way to move from a hardware-in-the-loop approach to a fully emulated environment is proposed by Zaddach [44]. Hereby, peripheral fingerprints are first generated by recording MMIO traces in a manner similar to PRETENDER. Afterwards, those fingerprints are used to match peripherals which are already *known* to a given emulator. While this allows accurate emulation of peripherals if successful, a single *unknown* peripheral would spoil hardware-less emulation.

Symbolic execution is even more impacted by such approaches; analyses using hardware-in-the-loop must be careful to only execute portions of code that do not contain hardware interactions, to avoid corrupting the hardware's state visible by all parallel code paths being explored. Cost also becomes a factor, as each analyst wishing to explore a set of devices must purchase and instrument the devices, which raises the barrier to entry for firmware analysis. While hardware-in-the-loop techniques do allow for interactive, relatively low-effort analyses, they are by no means adequate for thorough program analyses of arbitrary firmware.

A recent system proposed by Talebi, et al. [45] employs a hybrid of Linux-based rehosting and hardware-in-the-loop to enable the fuzzing of device drivers. While moving the bulk of the execution to an already-rehosted system presents speed improvements over

on-device fuzzing, it inherits the limitations of both approaches: dependence on the Linux abstraction, scalability limitations due to the need for a device, and incompatibilities with DMA.

Interactive. A re-hosting solution should be responsive to new program input. While defining the notion of input on an embedded firmware is itself a nuanced problem, the remaining hardware (not used as the source of input) should react accordingly. Trace replay-based solutions, such as PANDA [46], while quite flexible and useful for certain analyses, are not interactive and cannot be used to implement fuzzing or symbolic execution, which rely on this primitive.

Automatic. An ideal re-hosting solution should not require a significant effort per-device to use. The diversity in on-chip and external peripherals is so severe, that it is highly unlikely that any firmware can be emulated out-of-the-box with a commercial or open-source emulation package.

While some commercial systems provide the ability to rehost completely custom hardware architectures (e.g., Simics [10]), these systems still require the hardware models to be programmed manually. This is made worse by customizable CPU cores, and the diverse array of electronics components that the electronics industry continues to support. Even static and symbolic analysis tools [47, 28, 48] heavily rely on the manual specification of hardware behavior, particularly around IO and interrupts.

While there is little useful data able to quantify embedded CPU diversity, and documentation from vendors is not in a comparable form, we managed to locate a dataset of 555 CMSIS System View Description (SVD) files [49], which are XML files describing chipsets based on Cortex-M microcontrollers. They detail the on-chip peripheral locations and layouts of 463 distinct chips across 13 different chip vendors. This collection is by no means complete (it does not even include all of the chips used in our experiments

in Chapter 3 Section 3.2), but it shows the complexity and the scale of this problem. In this dataset alone, we could identify 1592 unique implementations of peripherals demonstrating the immense variety of peripheral and chip designs.

This complexity increases even more when considering external peripherals connected to the chip via on-chip buses and interrupt controllers. Hence, emulators such as QEMU [50] have to include carefully and—up to now—manually crafted implementations of peripherals and align them at the right location. In fact, the upstream version of QEMU only exposes implementations for three different Cortex-M chips, none of them present in the above dataset.

As a result, analysts end up creating their own peripheral and board implementations and maintaining them in separate forks of the project, such as *QEMU STM32* [51] or *GNU MCU Eclipse* [52]. A different approach is taken by *LuaQEMU* [53] and *avatar²* [34], which provide an interface for the analyst to define the peripheral layout. While these may be preferable to languages such as C used by QEMU itself, the analyst is still required to obtain and understand the full documentation for the particular CPU model used, and this effort may not transfer entirely to other similar CPUs, even from the same vendor. Therefore, it is very clear that an automated solution is needed to be able to make firmware analysis tractable.

In Chapter 3, we will present the first system aiming at ideal rehosting. Subsequent to this work's publication, other attempts have been made, such as P2IM [35]. Naturally, no system presented here or elsewhere claims to have completely achieved ideal re-hosting; we discuss the limitations of this in Section 3.3

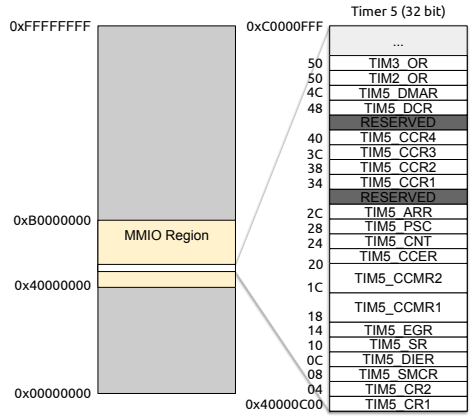

Figure 2.1: The memory layout for a simple 32 bit memory-mapped timer on the STM32 embedded processor.

2.3 Library Matching

HALucinator's LibMatch component builds upon related work in binary library matching and naming.

Previous work has explored various aspects of "function identification". As this term has many over-loaded uses, it is important to distinguish the problem LibMatch solves (labeling specific binary function names in firmware samples) from others. *BinDiff* [54, 55], and its open source counterpart *Diaphora* [56] use graph-matching techniques to effectively and efficiently compare two programs. While these tools can be effectively used to label functions, by matching a target binary to each library object, the tool does not account for collisions.

ByteWeight [57] identifies the locations (*e.g.,* start and end) of functions by computing a prefix-tree of likely sequences based on a database of known libraries but does not attach labels to located functions. LibMatch must also perform this analysis to eventually label function names in firmware, but ByteWeight itself does not attach any label to the functions it locates. The `angr` platform underlying LibMatch incorporates the results of

ByteWeight in its CFG recovery algorithm.

Multiple previous works have explored the problem of function labeling, using various combinations of features extracted from functions, and matching methods, to associate one set of code from another. Feature extraction techniques include function preamble-based signatures [58], backward slices from system calls [59], and traces from symbolic execution [60, 61]. Matching the extracted features has been performed through Bayesian networks [62], neural networks [63], and locality-sensitive hashing [64]. Unfortunately, none of these systems are suited for labeling functions in firmware due to several challenges: the inability to analyze or execute ARM Cortex-M code, the lack of information available to machine learning approaches due to small size and close similarity of functions in HALs, and the inability of some approaches to deal with collisions in an efficient way. This lack of existing approaches leads us to develop our function matching approach that is tailored to embedded firmware.

2.4 Binary Patching

Many previous works have explored the area of patching binary programs. Each has various pros, cons, limitations, and assumptions, which must be compared to understand the current state of the field.

Wenzl, et al. [65] dissect this area of research, and define four common steps to binary rewriting: *Parsing*, *Analysis*, *Transformation*, and *Code Generation*. The work in this area varies in terms of how they handle each of these steps.

The largest class of this work concerns *reassemblable disassembly* [66, 22, 21], the notion of disassembling a binary completely into standard assembly code for the architecture in question, adjusting the code, and then simply assembling it again into the finished binary. This is a preferable technique for the analyst, as it makes the Trans-

formation and Code Generation step easy; assembly can be easily patched manually or automatically via the human-readable assembly code, and turned into a binary again with standard tools.

However, this work presents problems when it comes to tackling the rewriting of firmware. First, reassembly assumes that the firmware can be disassembled completely during the Parsing phase. This includes the inter-related problems of distinguishing code and data, finding function boundaries, distinguishing pointers from integers, and the resolution of indirect jumps. Unless we can re-host the firmware into an emulated environment, we are left with performing those tasks entirely statically. We also are dealing with monolithic firmware images, and therefore do not have an explicit memory map, symbols, or other metadata that can make these steps easier. Therefore, we cannot guarantee the completeness of the disassembly, control-flow recovery, or function identification. As an indicator of this challenge, the samples in Chapter 5 Section 5.4 have an average of 401 unresolved indirect jumps after `angr`'s resolution mechanisms (based on [67], with our real-world firmware samples having an order of magnitude more than vendor sample code. Since monolithic firmware is not position independent by nature, if we cannot locate every pointer, including those used to access code and data, and adjust them during reassembly, the firmware will not execute correctly.

On top of this, many aspects of these approaches are architecture-specific, with the majority of works focusing only on Intel x86 binaries [66, 22, 68, 69], or requiring x86 hardware extensions [70]. Some caveats related to ARM instruction set features that inhibit the Parsing and Analysis phases were addressed by Kim, *et al.* [23], but the approach still inherits the aforementioned severe limitations of reassembly.

Another emerging class of work revolves around transferring patches from one program to another. OSSPatch [71] works by leveraging source code of open-source projects used in the unpatched target binary to retrofit patches from their upstream sources. A

similar work [72] aims to transfer the patched portions of compiled binaries, without the source code. In our scenario, the patch has simply not been developed, and will not be developed by the manufacturer, therefore we cannot leverage either of these approaches.

Finally, while the above work has explored the Parsing, Analysis, and Transformation aspects of binary rewriting, little thought has thus far been given to the Code Generation portion. Unfortunately, this is where the unique challenges of embedded systems, and particularly safety-hardened systems, begin to restrict our ability to patch. First, all of the above work assumes that, when a patched binary is created, the system will simply execute this binary instead of the original. As we discuss in Chapter 5 Section 5.1, most commercial embedded systems contain self-checks to guard against accidental modification or corruption of the firmware (e.g., CRCs and checksums), or intentional manipulation by an attacker or user (e.g., digital signatures) which must be located, and bypassed, for a patch to work. On top of this, it is also assumed by reassembly-based approaches that we have the toolchain needed to create a functioning binary image. In the case of a monolithic firmware image, we do not know what format, if any, is present in the firmware, or what tools may have been used to create it.

Additionally, we need to find space for our added code, either within the image, or by appending it to the end. Since we often obtain monolithic firmware in the form of full flash images (e.g., we cannot simply append data to the end), we must decide what in this image can be removed, or what apparently-empty space is indeed available. This relates to the problem of *binary de-bloating*, the known-undecidable problem of removing unnecessary content from a program. Recent work has proposed solutions for debloating Dockerized applications [73], during program compilation [74], or using shared library files [75]. Naturally, these kinds of approaches involve extensive knowledge of the program, in the form of source code, object code, or simply do not apply to the firmware domain.

Razor [76] uses a series of test-cases, along with the collection of execution traces and heuristics to determine removable portions of code. Unfortunately, as we mention in Chapter 5 Section 5.1, this kind of trace collection is not possible in the embedded systems domain, without a successful re-hosting solution. Redini, et al. [67] propose BinTrimmer, which aims to remove unnecessary code, without using any of the above assumptions. It does so through improvements to indirect jump resolution, which are used as the basis for `angr`'s own indirect jump resolver, although, as mentioned above, this too is not complete enough to be useful in this context.

In summary, these numerous issues combine to rule out any *Dynamic* or *Full Translation* approaches (as defined by Wenzl, et al. [65]), and we must resort to the more simplistic *Direct* or *Minimal Invasive* techniques when dealing with monolithic firmware images. Even these (such as Detours [24]) require us to solve at least the problems of code insertion, self-checks, and sources of security-relevant data, which we focus on in this work.

Chapter 3

Toward Firmware Analysis through Automated Re-Hosting

In this chapter, we propose an approach, aiming to achieve ideal re-hosting (as defined in Chapter 2, and propose a proof-of-concept system, called PRETENDER, which is able to observe hardware-firmware interactions and create models of hardware peripherals automatically. Our system first creates a recording of real interactions between the firmware and its hardware, and uses machine learning and pattern recognition techniques to create models for each peripheral on the CPU. The generated models can then be leveraged by popular full-system emulators (e.g., QEMU [50]) or program analysis engines (e.g., `angr` [26]) to enable precise, scalable, interactive analyses of the accompanying firmware.

While automated re-hosting may seem conceptually straightforward, the challenges in modeling even simple hardware-firmware interactions are numerous. We may think of a peripheral, such as a serial port, as a simple object that sends and receives data, but the firmware's view of this hardware is much more complex, consisting of dozens of individual configuration, status, or data registers, which, from the point-of-view of the firmware, appear as only opaque memory accesses, without any indication of their layout or behavior. Two peripherals performing the same function on two different CPUs,

even from the same vendor, vary wildly in terms of memory layout and implementation details. On top of this, accesses to these peripherals occur within the CPU itself, and obtaining these interactions for modeling is its own challenge. Interrupts are also a common feature of embedded peripherals, and must occur exactly as expected, or the hardware or firmware may fail.

To evaluate our approach, we demonstrate our recording and modeling techniques on a set of six unique "blob" firmware samples,each on three different hardware platforms, with associated external peripheral devices. Our experiments show that PRETENDER is able to successfully extract the peripheral models and execute the firmware in a fully emulated environment. The models offer enough interactivity to allow for the exploration of parts of the program not seen during recording or training. We further show the potential for direct applications to dynamic analysis, by using these modeled environments to trigger synthetic security vulnerabilities in the firmware samples. The hardware modeled in these experiments represents CPUs and other components common to low-power IoT and embedded devices. However, many challenges remain before typical commercial devices can be modeled in full. We nevertheless believe that the goal of automated firmware re-hosting is both achievable and necessary. Therefore, we conclude with a discussion of limitations, open problems, and next steps toward tackling the complexity of commercial devices.

In summary, our contributions in this chapter are as follows:

- We explore the problem of *firmware re-hosting*, and show that virtual, interactive, automatic, and abstraction-less approaches are needed to handle today's diverse firmware.

- We present PRETENDER, a proof-of-concept system able to automatically build hardware models, through a mix of novel hardware and interrupt recording tech-

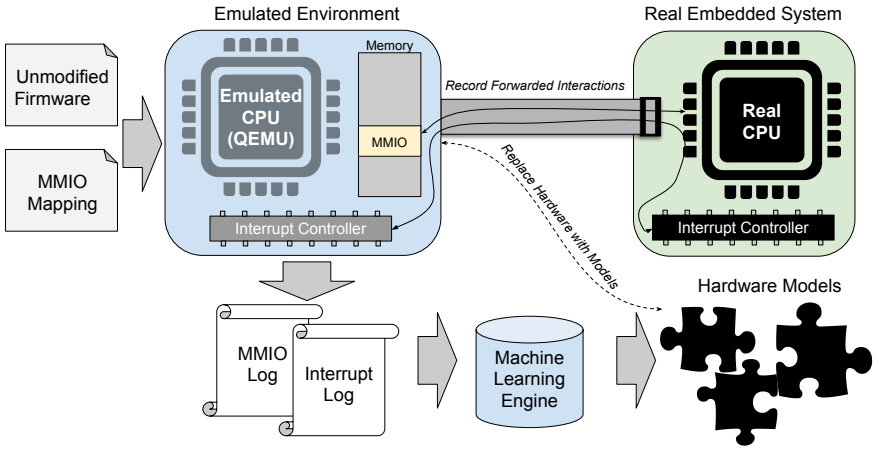

Figure 3.1: Overview of the functionality of PRETENDER

niques, machine learning, and peripheral state approximation.[1].

- We apply PRETENDER to multiple firmware samples across multiple hardware platforms and show that the generated peripheral models are accurate, automatic, and interactive enough to enable program analysis and vulnerability discovery.

3.1 Methodology

In this section, we present PRETENDER, a step toward automating the modeling of MMIO and interrupt-driven hardware peripherals to enable re-hosting. The goal is to gather data on, and build models of, these peripherals, such that the firmware under analysis can later be independently executed in a CPU emulator. We present our solution in the context of its use to support dynamic analysis of firmware, although the generated models have other possible uses, which we will discuss in more detail in Section 3.3.

The success metric we adopt to evaluate the completeness of the extracted models is what we call *survivable execution*, which we define as the ability for the firmware to

[1]To allow the reproducibility of this work, the source code to this work is available at `https://github.com/ucsb-seclab/pretender`

execute the same regions of code as it would if the original hardware were present, without faulting, stalling, or otherwise impeding this process. We include in this definition the need for our program to be interactive, as this is a requirement for many analyses. That is, the firmware and our hardware models need to be able to *operate on inputs and execute code paths that were not observed during the recording and model-generation phase*.

Assumptions and Prerequisites. We make a few basic assumptions in the implementation of PRETENDER.

- We assume that a CPU emulator is available for the target device, and that this emulator supports all CPU features that can impact control flow, including the interrupt controller.
- We assume the analyst has the ability to observe memory accesses and the occurrence of interrupts in the device in real-time. We will present a method for accomplishing this on any device with a basic debugging interface, lowering the requirement to the ability to read and write the device's memory.
- We assume that the basic memory layout of the target device is known, particularly the location of code and data in the memory space. More generally, we need to know where these areas are *not* located, as we can assume that the remaining areas are interesting locations we wish to model, including the MMIO regions.
- We assume that a human or automated process is able to interact with the hardware and that it achieves sufficient code coverage during the recording phase to reveal enough hardware interactions to generate a model. The more complete the code coverage is, the more detailed the extracted model will be.

A discussion of these assumptions can be found in Section 3.3.

PRETENDER works in the following phases:

1. **Recording.** We instrument the device to obtain a trace of accesses to the MMIO regions, and any interrupt that occurs during the execution.

2. **Peripheral Clustering.** We locate the boundaries of each distinct peripheral within the device's memory space, and divide the recording into sub-recordings for each peripheral.

3. **Interrupt Inference.** Based on the interleaving of interrupts with MMIO, we assign each numbered interrupt event to a peripheral group. We then infer which bits in which memory location in the peripheral control interrupts, and create timing patterns to be used during emulation.

4. **Memory Model Training.** In this step, we attempt to select and train known models for each memory location within the identified peripheral regions. Any unidentified memory locations will be modeled using State Approximation.

5. **Test Harness Creation.** Finally, the analyst must decide how input should be introduced into the system, through the creation of a simple test harness. This is the only manual step in the process, as the decision depends on the analyst's needs.

A complete overview of PRETENDER and the interplay between its different parts can be seen in Figure 3.1. In the remainder of this section, we will discuss the individual phases of the system in detail.

3.1.1 Recording MMIO

The natural first step in building models of hardware is recording a trace of the IO activity that occurred during execution. As we outline in Section 2.2, the firmware depends on both internal "on-chip" peripherals, and external "off-chip" peripherals, both of which are needed for the firmware to operate as expected. However, the firmware only

communicates with off-chip peripherals through its interactions with on-chip peripherals, so in order to have a complete recording, we must capture all memory accesses that constitute MMIO.

Rationale. Peripherals are considered "memory-mapped" because they are attached to, and addressed via, one of the CPU's internal memory buses. Unlike external buses, which can be physically probed and monitored, these interactions only occur within the CPU's die, and cannot be directly monitored. While some debugging facilities used in the development of new chips offer a data trace of the memory bus, such as ARM's ETM/HTM Data Trace, these features are seldom available on production chips, and are entirely absent in the low-cost, low-pin-count chips of commercial embedded devices. Typical CPUs found in the wild include, at best, a debugger capable of simple execution control, and memory/register access.

On top of this, MMIO behaves differently from a normal region of memory; instead of just storing data, these locations instead control or represent aspects of on-chip peripherals. Their value or function may change based on external factors, without any interaction with the firmware.

One possible alternative approach to MMIO recording would be to instrument the firmware to record IO interactions. This requires us to understand, from the binary firmware itself, where this IO takes place. This could be done on architectures where explicit `in` and `out` instructions are used for peripherals. On ARM, however, this is not a straightforward operation, as peripherals are accessed via normal memory handling instructions (`LDR`/`STR`), and it is often difficult to tell statically whether an instruction is addressing a peripheral or normal memory. Inserting this instrumentation code non-destructively, and collecting the cumbersome volumes of data it generates, are both hard problems, and may even be impossible if the code is present on a Read-Only Memory (ROM).

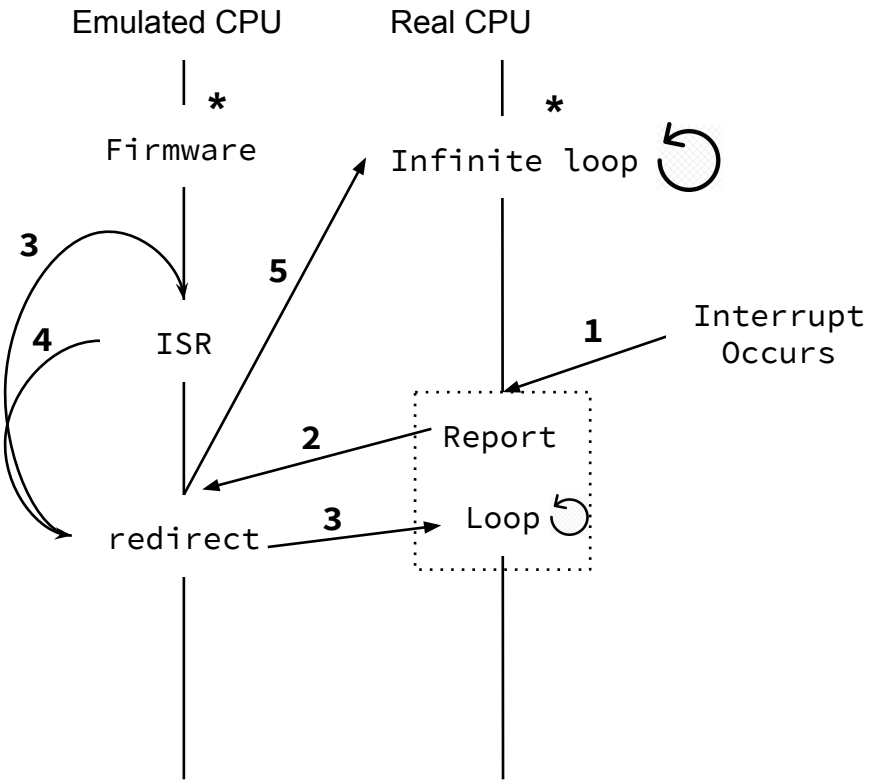

Figure 3.2: State diagram of interrupt recording in PRETENDER. * indicates the initial state.

Recording Implementation. As a result of these complications, our recording approach leverages a hardware-in-the-loop execution approach, where the firmware is deployed in an emulator, and the MMIO requests are forwarded to the original hardware, which allows recording in-transit. We built upon the *avatar*^{2} framework [34], which allows for the simultaneous control and orchestration of emulators and hardware. *Avatar*^{2} supports an event-based callback infrastructure, which allowed us to implement the recording of memory events.

3.1.2 Recording Interrupts

In order to fully model on-chip hardware peripherals, we must observe the interrupts that they generate, *in the context of the MMIO activity of the firmware.*

Rationale. Interrupts play an important role in most peripherals, and are a particularly difficult aspect to record and model correctly. Interrupts are triggered by some event, whether it is an explicit MMIO operation, or an event in the physical world, and cause the execution of Interrupt Service Routines (ISRs) as a result. These ISRs typically contain MMIO operations associated with the peripheral that triggered the interrupt (e.g., reading data that arrives at a serial port or counting the number of times a counter overflows). Without the peripherals' ISRs executing at the correct times, the peripherals may not function, or the system may crash. This behavior is a property of the hardware itself; the internal logic of the peripheral decides when and how often to trigger its associated interrupts. Many peripherals allow this behavior to be adjusted at runtime, through their configuration registers. For example, many peripherals have a single bit in their configuration register controlling whether interrupt events are generated at all.

Hardware features exist on many chips for providing a log of the interrupts, such as ARM's Instrumentation Trace Macrocell (ITM), but these features are not universal, and are difficult to coordinate with simultaneous peripheral recording or even basic hardware-in-the-loop emulation. Hence, previous solutions, such as the first version of the Avatar framework [29] or SURROGATES [32] tried to tackle interrupt forwarding with custom stubs injected onto the device under analysis. However, both of these solutions forward interrupts in a "fire-and-forget" manner. This results in inconsistencies between hardware and emulated firmware, as incoming interrupts on the hardware could easily be missed when the emulator serves a previous interrupt. Although those inconsistencies are a negligible problem for manual analysis, they dramatically complicate automated modeling,

and must be avoided. A more recent approach, presented by Corteggiani et al. [42], uses a custom tailored protocol to keep hardware and emulator *synchronized* during interrupt forwarding. Unfortunately, this method requires custom debugging hardware that would greatly reduce the generality of PRETENDER.

Hence, we heavily extended *avatar²* to support the notion of forwarding and recording interrupts, while carefully keeping the two systems synchronized without the need of specialized debugging hardware. The current published version of *avatar²* retains the hardware in a "debug-halt" state while forwarding memory accesses, in order to avoid side-effects from the resident code. Unfortunately, this debug-halt state inhibits all interrupts, and thus cannot be used as-is. However, we cannot simply keep the CPU running and forward all of the generated interrupts into the emulator; if too many un-handled interrupts arrive, or spurious, unwanted interrupts occur, the hardware or emulator can experience an unrecoverable fault. The current version of *avatar²* also does not support writing to memory while the CPU is running. To make matters worse, halting the CPU during interrupt routines is problematic, as we noticed that some peripherals, particularly those that control future interrupts, will not work properly in this halted state because they are bound to the CPU's instruction pipeline. As a final complication, we must ensure that we return from these interrupts properly, both in the emulator and on the hardware to ensure that the hardware continues to function, even though it is not executing any code.

Interrupt Recording Implementation. Figure 3.2 shows how interrupts are recorded in PRETENDER. As interrupts are generated on the real device, we should have the Real CPU running. Hence, we always have the Real CPU execute an infinite loop. Furthermore, we replace the ISR of all the interrupts with a recording stub (shown in dotted box in the Figure 3.2).

When an interrupt occurs (Step 1), the recording stub is triggered, which immediately

reports the interrupt number to PRETENDER (i.e., the Emulated CPU), and halts the Real CPU (Step 2). The emulated CPU then starts executing the actual ISR for the corresponding interrupt, and directs the real CPU to run a loop in the interrupt's context to mimic the execution of the interrupt (Step 3). Once the ISR completes execution on the emulated CPU (Step 4), PRETENDER redirects the execution of the Real CPU to the default infinite loop, and the Emulated CPU to continue executing the firmware (Step 5). This ensures that both the hardware and emulated interrupt controllers are synchronized.

3.1.3 Peripheral Clustering

With the combined MMIO and interrupt recording collected, we can now proceed to reason about and model the peripherals themselves. In the end, we need to construct a model, such that each MMIO location that the firmware accesses returns a reasonable value. However, these locations are not independent; multiple locations represent one logical device in the silicon of the chip, which has its own concept of state, control interrupts, and so on. For example, writing a byte to the data register of a serial port may cause the "transfer in progress" or "busy" flag to become active in the same peripheral's status register. Therefore, a major prerequisite to the future modeling steps is to group all memory accesses by their associated peripherals.

To do this, we rely on the intuition that each MMIO peripheral is typically associated with a block of contiguous memory addresses (e.g., `0xC00-0xCFF` in Figure 2.1) . While we cannot be sure exactly what the boundaries between the peripherals are, we assume there is some fixed alignment for—and the minimal gap between—them, likely due to the underlying details of the peripheral buses that serve MMIO peripherals. These details are supported by the SVD data explored in Section 2.2, as well as the manuals for all of the devices explored in Section 3.2. We can, therefore, find our peripheral boundaries through

clustering techniques. For this work, we take the set of accessed addresses and employ the Density-based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [77] to recover the peripheral groupings.

The intuition behind this choice is that each peripheral will appear as a small cluster of accesses in a relatively sparse memory space. For example, in Figure 2.1, while an entire page of memory (`0x1000`) is allocated to the timer, only a small portion (`0x00-0x50`) of that memory space is actually used, meaning that subsequent peripherals in memory will likely have large gaps between their relative clusters. DBSCAN is able to quickly discern these clusters, providing us with the capability to efficiently group the various accesses. In our work, we set our maximum gap between any of the addresses in a cluster (i.e., *epsilon*) to be `0x100` and the minimum cluster size to be *one*. Almost any reasonable value for epsilon (e.g., `0x8-0x100`) would likely produce identical and useful clusters, and our minimum cluster size of one ensures that we will not exclude simple or infrequently-used peripherals from our models.

3.1.4 Interrupt Inference

In order to model interrupts correctly, we need to establish a reasonable approximation for when to fire each interrupt and which MMIO event triggered it. First, we find the association between the interrupt number and the peripheral firing the interrupt, which is a property of the hardware that varies widely between chip models. Then, we discern which MMIO register is used to enable and disable each interrupt, so that we do not fire it too soon or too late in the execution. Finally, we determine how often to fire interrupts when they are eventually enabled.

To associate an interrupt with a peripheral, we examine the interleaved interrupt and MMIO traces and locate all of the MMIO operations that occur during an Interrupt

Service Routine (ISR) (e.g., between an interrupt event and the emulator returning from the ISR). We leverage the intuition that the purpose of most interrupts is to trigger the firmware to communicate with the interrupting peripheral, by executing the code in the ISR. Therefore, we associate an interrupt number with a peripheral if that peripheral's MMIO addresses were accessed the most during the ISR's execution.

We then locate the memory location containing the interrupt's *trigger*, which is a location in the peripheral which, when a certain bit pattern is written, causes interrupts to be enabled. The location can be determined by finding the very first interrupt for a given interrupt number, and seeking backward in the MMIO/interrupt trace until a write to the associated peripheral is found. This is intuitively the configuration, or interrupt-enable register, as it is best practice to enable interrupts as the final step during peripheral configuration, as, after this point, any operation could be interrupted. However, this memory location may be shared with other functions, and many bit patterns may be written to it during an execution which have no effect on interrupts. The next step is therefore to refine the bit pattern which can enable interrupts in the model, based on which writes appear to control interrupt behavior in the hardware. We start with the assumption that all bits in the trigger location control the interrupts. For each write to the detected trigger location, if a bit is set to 0 when interrupts occur, it is unlikely to be the interrupt trigger bit, and is removed from consideration. The remaining bits are considered the final interrupt trigger; during emulation, when these bits are set in the trigger location, interrupt events will be fired by the model.

Finally, we must determine how often to fire interrupts when they are enabled. There are various kinds of interrupts: *pulse* interrupts occur once for every event they represent, and *level* interrupts occur repeatedly until some MMIO action disables them. While level interrupts would be easy to model based on the state of the peripheral, we cannot reliably distinguish these two types in the recording data. As a result, the most general, flexible

approach is to use interrupt timings. Interrupts can also be very frequent. Since these are the timings seen during PRETENDER's recording, we can be sure that the emulator can at least support interrupts at this speed. We collect the timings between an interrupt return and the beginning of the next interrupt (as well as between the trigger and the first interrupt) and create a repeating sequence. As long as interrupts are enabled via the correct bits in the interrupt trigger location, they will be fired repeatedly until they are disabled.

The result is a peripheral model for which interrupts can be enabled and disabled by the program in a realistic manner, and with timing intervals that the emulator can support. We find that these intuitive heuristics both align well with the design of peripherals, and also work well in practice, as we show in Section 3.2.

3.1.5 Memory Model Training

In this step, we select a model for each memory location in a peripheral. We first look for common memory access patterns, which allow us to train accurate models for these common types of interactions. For some memory locations, where more complex, stateful, functionality is implemented, we employ a *state approximation* mechanism, able to provide known-valid sequences of observed values for that specific memory location, based on what state we infer the peripheral to be in.

There are a few basic types of MMIO registers common to many peripherals (e.g., configuration registers, status registers, and counters). By using simplified models for these, we can allow this part of our model to maintain flexibility, and operate as independently as possible from the circumstances of the recording. We identify and model a number of classes of MMIO:

- the *Simple Storage Model* is used for memory locations that were observed to *always*

act like normal memory. That is, the value returned for a read from a location was always identical to the most recent value written to that location;

- the *Pattern Model* is used for memory locations whose read values appear to follow some repeating pattern (e.g., $0, 1, 1, 0, 1, 1, ...$), including locations that always return a static value;

- the *Increasing Model* is used for values that are *eventually* monotonically increasing (i.e., the last half of the observations were increasing), which is typically indicative of a timer or counter;

- and the *Write-only Model* is used for memory locations that were only ever observed to be written to, which are effectively ignored from a modeling perspective, but interesting for our state approximation, as they are likely configuration registers that directly affect the state of the peripheral.

While these models are relatively straightforward, our Increasing Model requires multiple iterations of linear regression modeling to find the best fit line. This is because these incrementing values are typically configured during the boot process, which means that their initially read values are unlikely to be indicative of the actual rate of increase. For example, a counter may start on boot at a certain rate, then the firmware will configure a new rate and reset the timer, resulting in two distinct functions represented by the same memory value. To handle this, we iteratively remove outliers (i.e., values that have a correct p-value greater than 0.0001) from our regression model until we have a good-fitting function for the steady-state increase. When we are replaying this model, we first replay the initial outlier values verbatim, and only switch our projection function once initial values are exhausted and the long-term behavior is expected.

State Approximation. The remainder of locations within a peripheral represent those

locations that do not follow any easily identifiable pattern. These locations can represent external sources of input or external physical phenomena, reflect large amounts of state invisible to the CPU (e.g., the internals of on-chip peripherals), and be related to the behavior of interrupts. Therefore, methods relying on function-fitting or direct recovery of a state machine involving these memory locations simply will not suffice.

Rationale. Our state approximation model is used when a MMIO location does not fit any other model. According to our observations, these tend to be the locations in a peripheral directly affected by external events, such as the data register of a serial port, a bus controller, or a status and event flag register.

These locations are the most challenging to model and emulate. For example, in the case of an I2C bus controller, there are many sources of state, and numerous causes for the state to change, many of which are not observable. From the software's perspective, the I2C bus controller presents an MMIO interface, which specifies how the bus protocol is spoken (baud rate, master/slave), whether queuing is enabled or interrupt are fired, and so on. At another layer, the hardware between the MMIO and the pins has a state, containing the data queue, bus-related timers, and other condition flags not visible directly through MMIO. Both of these portions also occur in the device on the other side of the bus. Finally, the two devices share a protocol spoken on the I2C bus itself, which specifies an ordering of events (start symbol, address, data with acknowledgment, etc.). The result of this is a series of composed, inter-related state machines, which also rely somewhat on the physical world's events, and can only be observed through the rather limited window of MMIO memory accesses.

Unfortunately, this means that we fail the requirements of state machine recovery techniques, which are typically used to infer states and transitions from an activity trace. We do not know the number of possible states, we cannot tell when two states are equivalent, and it is challenging to know concretely if we have even changed the state

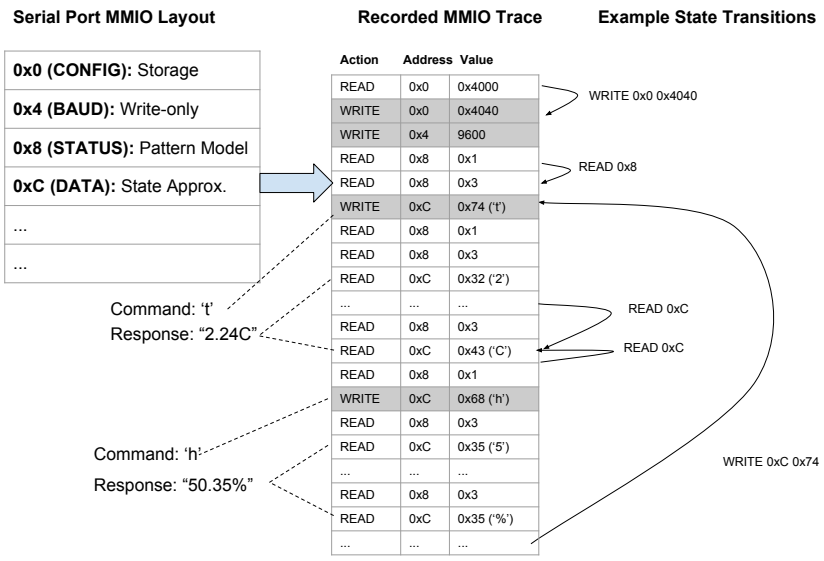

Figure 3.1.1: Illustration of State Approximation in action, on a simplified serial port peripheral

of the peripheral. We also cannot easily distinguish data registers, which may contain data respecting some protocol, from others containing status flags, error codes, and configuration data. However, it is also not sufficient to simply replay values verbatim from the recorded trace. This is because our models need to be able to function even when we observe deviation from the recording caused by new input, timing-related deviations caused by differences between the hardware and emulator, as well as to tolerate the asynchronous and non-deterministic occurrence of interrupts.

State Approximation Implementation. As a first step toward addressing these challenges, we instead make an approximation of the device's state, using only the observed trace's data and ordering, by inferring state transitions we know must exist. We observe that writes to MMIO addresses are typically used to cause a change in state (e.g., the transmission of data to external hardware or a change in the internal configuration of a peripheral), and approximate that the activity between two writes found in an MMIO

recording may roughly represent the same state of the overall peripheral. Interrupts also represent a change in state, although we cannot know concretely what change in state they represent. Reading data can also change the state of a peripheral, but in a more subtle way (e.g., reading a byte from a serial port causes it to be removed from an internal hardware buffer, and a subsequent read to the same address will return a different value).

With these intuitions in mind, our State Approximation model consists of the trace of MMIO and interrupt activity for a given peripheral, and a *state pointer* consisting of where in the trace we believe best approximates the state of both the program and the peripheral. At the beginning of execution, the state points to the beginning of the trace. We update this state based on the following rules: When an MMIO address for this peripheral is read, we look ahead in the trace to find the next time this location was read. If it is found, we return this value, and update the state pointer to this location. If we encounter a write, an interrupt, or the end of the trace before we find one, we instead return the most recent value for that location, and do not update the state pointer. This encodes the behavior that values read from MMIO may be sequential (as in the serial port buffer mentioned earlier) and that they respect the boundaries of state caused by writes and interrupts.

When a write to the peripheral's MMIO occurs, or the associated interrupt event is triggered, we look forward in the trace for the next location where the same event occurred, and update the state pointer. If we do not find it before the end of the trace, we instead seek backward through the trace. If the value written is entirely new, we do not update the state pointer. These rules allow our model to respond intelligently to changes in its mode, or new commands, regardless of the order they occur during execution, particularly when new input causes deviation from the trace.

State Approximation Example. As an example, consider a hypothetical device that

uses a serial port to act as a client for the thermostat we model in Section 3.2. This device's firmware will query the thermostat, with 't' and 'h', and expect a properly formatted temperature or humidity in return. Furthermore, the firmware reacts to this data, for instance by sending the information across a network, or raising an alarm. The device firmware must receive a response from the thermostat when expected, and the response must make sense for the given command, for the firmware to behave correctly.

An illustration of what this model might look like can be seen in Figure 3.1.1. Note that, in a real-world scenario, there will be many peripherals needed to operate the firmware, but here we focus on just one to better explain its behavior. The client device's serial controller contains many registers, including a configuration register, a status register, a data register, as well as assorted registers governing physical hardware details, like baud rate. Each of these is addressed by its own MMIO location, in a contiguous memory region we identified during clustering. We notice, from our traces and previous Memory Model Training, that the configuration register is a simple storage location, and the baud rate control register is only ever written to. The contents of the status register follow a pattern, alternating between the values `0x1` and `0x3`, which we will interpret as whether data is ready to receive or not. The data register, on the other hand, will change without respecting any pattern or direct stimulation from the firmware. Therefore, this location is handled by State Approximation.

When emulation begins, we start in the peripheral's initial state; during boot-up, the firmware configures the serial port, writing to the configuration register to enable the serial port, and set the baud rate to 9600, advancing the peripheral's state pointer to the point at which these actions occurred. The firmware then begins its main loop, and requests a temperature, by writing a 't' into the data register. Naturally, the next thing that happens chronologically is for the status register to indicate that bytes are ready to read, and the firmware will read a temperature value out of the data register one byte at

a time (e.g., "24.24C"). Similar actions occur if an 'h' is written to the data register by the firmware; the status register indicates new data, and the firmware reads it back (e.g., "50.35%"). However, when emulating with new input, interrupts, or after the duration of the original peripheral's chronologically observed states, we must make a decision about what state the peripheral is in. In these cases, following the simple rules in Section 3.1, we will enter the state where a 't' or an 'h' was written to the data register, and subsequent reads will return a temperature or a humidity. In this simple example, the serial port will, after some time, return only the last valid temperature and humidity values, but it will continue to return only temperatures or humidities when asked for, and respect whatever formatting or encoding for these responses the thermostat uses, which may be checked by the firmware.

Test Harness Creation. Finally, in order for this system to be fully *interactive*, as we discuss in Section 2.2, the analyst must decide how input is to be introduced into the emulated environment. No standards exist for input and output in embedded firmware and hardware; exactly where an input is introduced is both a function of the target device's hardware, and the analyst's goals. For example, a serial port, in one device, could be connected to a human-controlled terminal (the obvious source of input), while in another, it could be wired across the circuit board to a simple sensor with a serial interface (a model-able device). PRETENDER, therefore, requires the analyst to provide their own means of input, in the form of a test harness. We leverage *avatar²*'s Python scripting interface to allow any MMIO location to be easily replaced by custom logic. As an example, for the firmware presented in Section 3.2, we created a harness consisting of feeding input data via the device's serial port.

3.2 Evaluation

To demonstrate the efficacy of PRETENDER, we use it to create models of the hardware in the context of multiple firmware images. We then use these models, together with freshly generated inputs, to uncover code paths and orderings *not seen during recording and modeling.* The newly covered parts of the firmware include synthetic security vulnerabilities, which the system is able to trigger and detect within the modeled environment.

Targets. We applied our system to firmware running on three different embedded CPUs on development hardware, the ST Nucleo L152RE, the Maxim MAX32600MBED [78] and the STM Nucleo F072RB [79]. The targets represent ARM-based microcontrollers common to embedded applications; the first two represent Cortex-M3-based designs, while the latter is based on a Cortex-M0. The layout of the peripherals, and the function of each MMIO register varies widely, even between the two targets from the same vendor. It is worth noting that QEMU has no official support for any of these chips, or any of their contained peripherals. Third-party forks contain partial support for related chips but would have to be heavily adapted and extended to work on these firmware samples. Access to all devices was obtained using a commodity CMSIS-DAP debugger. We showcase the function of our models in-depth in the context of the STM Nucleo L152RE, but provide results from all three.

We evaluated our technique on six example firmware: four of these were directly obtained from the ARM `mbed` [80] development suite's library of examples. These were designed to exercise interesting features of the hardware, and we chose them to demonstrate the challenges PRETENDER has to overcome for successful hardware modeling. We extended three of these examples with additional functionality, which we do not trigger during the recording and modeling phases. Besides additional hardware interactions, our additions also include synthetic security vulnerabilities, similar to the kind that an ana-

lyst may wish to locate in a binary firmware. The other two examples, not taken from the `mbed` examples, are more complex and mimic real-world firmware found on a door lock controller and a thermostat. All of our examples were compiled using GCC 5.0, and ARM's `mbed` hardware abstraction layer. While we had the source code available during our analysis, it should be noted that no part of PRETENDER leverages this information; PRETENDER operates solely on binary firmware and the hardware itself. While this may seem like a small number of samples in comparison to previous approaches [33, 36], the need to obtain and instrument original hardware necessarily limits the number of firmware samples.

We evaluated our system's effectiveness in terms of its achieved code coverage on each example, as measured through execution traces from QEMU. We note that good code coverage during our recording phase is an important factor in our modeling, as we want to explore as much of the hardware's functionality as possible. Table 3.1 summarizes the used peripherals and execution behavior of each firmware. We note that the reported block counts are approximate, particularly for those examples with interrupts, as QEMU re-defines basic blocks based on where an interrupt occurs and returns, leading to imprecision. The table shows vastly different amounts of covered basic blocks for the same firmware across different devices, although the exact same compiler, source code, and system library was used for all of the examples. This hints toward the many subtle differences in the hardware abstraction layer, which are required to deal with the diverse hardware platforms. The block count in the "*Rec.*" column serves for baseline comparison and shows the coverage reached during the initial recording phase. The "*Null Model*" column represents the coverage obtained when all MMIO is replaced with a model that simply returns a zero value for every location (this is in contrast to not having a model at all, where all of the firmware would cause QEMU to crash). The "*SA*" column shows the coverage with complete modeling, including the State Approximation of the firmware's

Table 3.1: Approximate basic block coverage for firmware samples with PRETENDER, as measured by QEMU

Firmware Name	**Peripherals**	**Blocks Executed**			
		Rec.	Null Model	SA	Fuzzing
Nucleo L152RE					
blink_led	Timer, GPIO	218	86	218	n/a
read_hyperterminal	Timer, GPIO, UART	545	85	545	636
i2c_master	Timer, I2C, AM3215	1185	61	1185	n/a
button_interrupt	Timer, GPIO, Button	344	68	314	n/a
thermostat *(custom)*	Timer, I2C, AM3215	1263	62	1261	1276
rf_door_lock *(custom)*	Timer, GPIO, Radio,	665	87	665	758
Nucleo F072RB					
blink_led	Timer, GPIO	405	117	405	n/a
read_hyperterminal	Timer, GPIO, UART	828	102	828	999
i2c_master	Timer, I2C, AM3215	1572	103	1572	n/a
button_interrupt	Timer, GPIO, Button	362	103	362	n/a
thermostat *(custom)*	Timer, I2C, AM3215	1662	103	1662	1918
rf_door_lock *(custom)*	Timer, GPIO, Radio,	960	102	960	972
MAX32600MBED					
blink_led	Timer, GPIO	280	9	280	n/a
read_hyperterminal	Timer, GPIO, UART	514	8	514	668
i2c_master	Timer, I2C, AM3215	941	8	942	n/a
button_interrupt	Timer, GPIO, Button	188	8	188	n/a
thermostat *(custom)*	Timer, I2C, AM3215	1009	8	1009	1066
rf_door_lock *(custom)*	Timer, GPIO, Radio,	692	8	692	712

source of input. A firmware that is entirely input-driven will have finite behavior when the source of input is modeled, but unlike previous approaches, the firmware will continue to execute after the input ends, but with no additional input-triggered behavior. We manually verified that all of the firmware samples performed the same overall behavior as was present during recording. That is, even when no hardware was present, the firmware used our generated models to function similarly to when it was running on the actual

hardware. In the last column, *Fuzzing*, we feed automatically generated random data to the three firmware examples whose execution is data-dependent, which is equivalent to a naïve fuzzing approach. We accomplished this by attaching a test harness in place of a serial port controller to the system, which, instead of supplying modeled data, provides IO from the host system. This allows new input to be supplied to the firmware program for exploring new functionality, while letting the rest of the PRETENDER-created models function normally. As the table shows, PRETENDER successfully discovered new blocks, and, subsequently, revealed new functionality of the firmware. In all cases, this extra functionality actively interacted with the *other* peripherals models, such as timers and system configuration, not just the serial port. While we discuss details of the hardware peripherals when commenting on PRETENDER's behavior, our system is not aware of the specific layout, names, or functionality of any of the peripherals, aside from the test harness, and basic details of the standardized interrupt controller coupled to the CPU.

Our evaluation demonstrates that PRETENDER is able to successfully allow re-hosting, while enabling *survivable execution* at the same time. As a result, analysis techniques such as fuzzing could be parallelized and scaled. Rather than simple random data, smarter fuzzing techniques [81] could be used; however, we would like to emphasize that the goal in this work is not specifically to find new bugs in firmware via fuzzing, but to enable dynamic analysis, which is necessary to achieve this, and other security goals going forward.

In the remainder of this section, we will describe the hardware platform and each example more in-depth, together with the detailed re-hosting capabilities enabled by PRETENDER.

blink_led. This simple example blinks a Light Emitting Diode (LED) every 0.5 seconds. While this example may seem overly trivial, we use it to illustrate the basic level of complexity inherent in any firmware compiled with ARM `mbed`, and the basic behavior

Table 3.2: Snippets from a capture of all memory-mapped input/output (MMIO) accesses from an STM32 firmware.

(a) Increasing read-only (Timer 5 @ 0x40000C24)

Op. #	Operation	Value
	...	
524	READ	3690781
	...	
595	READ	3731433
	...	
658	READ	*3534604*
662	READ	5549086
663	READ	6053877
665	READ	7060952

(b) Read/write storage (Flash controller configuration @ 0x40023C00)

Op. #	Operation	Value
	...	
14	READ	0
15	WRITE	4
16	READ	4
17	WRITE	6
	...	
77	READ	6
78	WRITE	7
79	READ	7

of timers. When booting even the simplest firmware, the board performs a number of initialization tasks, including using the Reset and Clock Control (RCC) to enable various clock devices, the management of the on-board flash controller, and the configuration of GPIO pins. The firmware performs various self-checks on these peripherals during boot, and if they fail to report correct status information, the firmware will hang in an infinite loop. While this can also be solved with simple replay, the ability to execute this firmware indefinitely can only be achieved using modeling. Table 3.2 shows a memory trace acquired by PRETENDER, and shows interactions with the timer (Table 3.2a) and the flash memory controller (Table 3.2b). PRETENDER correctly identified the timer as an `Increasing Model`, and our linear regression approach correctly resolved the rate at which the timer increases. Whenever `wait()` is called, the value of the timer is periodically checked and the firmware continues execution only when it exceeds an ever increasing amount. PRETENDER's model can correctly produce the required values indefinitely.

Furthermore, the various RCC and other system configuration registers checked by the timer and GPIO code continue to produce the correct values, as we correctly deduced their simplified storage, pattern, and state-approximated values.

read_hyperterminal. This firmware receives external input from a user or other device over a serial port, and turns an LED on or off ("1" or "0") based on the input. This example shows diverging firmware execution based on different inputs, as a user can send various possible inputs, in any order. We stimulated the program by sending random "on" and "off" commands over the serial port for the duration of the recording. During our State Approximation-based execution, we were able to identically reproduce the execution. After the recorded input ends, the firmware continued to execute, waiting for more data from the serial port. To make things more interesting, we added a special backdoor to the firmware code. More precisely, if a "2" is sent, the firmware will prompt for a password, a common behavior for a hidden backdoor functionality. This functionality is also vulnerable to a buffer overflow when reading the password. In order to explore code-paths of the program not seen during recording, we use the serial port test harness described above, and provide random bytes as input. Even though this backdoor was not exercised during our recording, PRETENDER was able to successfully rehost the firmware accurately enough so that our emulated version can handle this input, including the various timer and RCC interactions present in this section of code. When fuzzing the rehosted firmware, we were also able to trigger the implanted buffer overflow, leading to corruption of the program counter, and crashing the emulator.

button_interrupt. This example makes use of interrupts that are triggered by an external event (i.e., a physical button). When the physical button is pressed, it causes an interrupt to execute a callback that blinks an LED. During our recording, we pressed this button at random intervals over a period of two minutes. Our recording function-

ality receives the interrupt events and forwards them to the emulator, which in turn executed a callback that manipulated the GPIO peripheral. We located the trigger for the GPIO interrupt automatically (`0x40010408` with value `0x002000`). However, as the timings for the individual button presses were random, PRETENDER falls back to State Approximation for this peripheral, still allowing indefinite execution.

i2c_master. This example is modified from the original ARM `mbed` example to support an AM2315 I2C temperature sensor, and reports both the temperature and humidity in the room. Unlike the previous examples, this one contains multiple sources of interrupts; both the primary system timer (TIM5) and the I2C bus produce interrupts, which causes a conflict during recording. For this reason, we utilize the iterative modeling approach described in Section 3.1. On the first execution, we obtain a recording of the timer's overflow-related interrupts, and convert this into a model. On the second execution, PRETENDER identifies that we have an interrupt-enabled model of the timer already, and uses it instead of the hardware. With this source of interrupts removed from the hardware, we are able to clearly observe the I2C bus's interrupt patterns. This peripheral has multiple bits that control interrupts, and through observing the peripheral, we are able to locate the correct bit mask for the configuration register (`0x720`), such that these bits being enabled will cause our timing-based interrupts to occur. While this bus is a source of external input like our serial port, the input is only generated in response to an action by the firmware. Therefore, when the firmware writes the configuration and data registers for the I2C bus with the appropriate values to read from the temperature sensor, the state of the peripheral will advance or rewind to the appropriate time that this action occurred during recording and the events will occur as expected.

Thermostat. In this example, we present a firmware that would drive a typical thermostat, indicative of popular smart thermostats (e.g., Google's Nest). The firmware

reads the temperature and humidity from the AM2315 sensor used above, but now it also accepts commands that poll for the temperature and humidity. If the temperature is too far from a preset temperature, it will enable a GPIO to trigger a hypothetical air conditioning unit. However, in order to showcase that peripheral models generated with PRETENDER are not firmware-specific and can easily be transferred and reused, we did not actually leverage a recorded peripheral trace to build the models for this firmware.[2] Instead, we reuse the models from the i2c_master example above, together with our test harness to uncover new functionality offered by the firmware. However, when we fuzzed the firmware using our test harness, we were able to discover this previously un-reached functionality, which directly results into an increased coverage as shown in Table 3.1.

Rf_door_lock. This firmware uses a Grove Serial RF Pro radio module connected to an Universal Asynchronous Receiver/Transmitter (UART) peripheral, which accepts multiple commands. Among others, those commands include "ping" and "unlock," which accept a password. If the password is correct, the firmware activates a GPIO, which unlocks a hypothetical mechanical lock. The functionality of this firmware is indicative of those on popular IoT smart locks. The radio module operates over a standard serial port. It can be configured using various commands, and once this is complete, it will simply transmit data received on the configured channel to nearby radios. Similar to many small embedded systems, this firmware provides a binary protocol we can use to send commands via our hypothetical smart lock client, including unlock (`0xbb`) and ping (`0xdd`). To interact with this firmware during recording, we used another radio device to send random valid and invalid lock codes and pings to the firmware. This firmware has an additional functionality, implemented as a backdoor that allows any radio user to overwrite the lock code, by sending command `0xff`, followed by the desired code; this

[2]Note that we obtained a recording of the firmware's execution nevertheless to provide coverage information for comparison.

feature is also vulnerable to a buffer overflow. As our radio uses a normal serial port, State Approximation works as expected here, but we cannot directly apply our serial port model and feed it with random data to reach additional block coverage. Instead, we need to correctly format our inputs according to the format observed by the radio's responses during recording; it checks that the radio responds correctly with "OK" to configuration commands, and will halt execution if it does not. This would be an excellent starting point for a mutational fuzzer, but for the sake of simplicity, we simply "mutate" by appending random data to the end of the data held in our model, and replaying it into our serial port. With this rudimentary fuzzer, we were able to automatically discover the hidden functionality, and even trigger the bug, causing QEMU to halt the execution.

3.3 Discussion

We have shown that a virtual, interactive, and automatic re-hosting solution is necessary to tackle the diversity in IoT and embedded devices, and demonstrated the possibility of such a system through PRETENDER. However, we fully acknowledge that the problem of automated re-hosting is still challenging to be completely solved. This section discusses the assumptions and prerequisites laid out in Section 3.1, and explores a number of the open problems and challenges that must be overcome in order to apply re-hosting in any context to production embedded devices.

Beyond ARM and MMIO. Currently, PRETENDER supports ARM devices, for which an emulator for the instruction set and any core peripherals (those which control code execution directly) are available. This is a reasonable requirement, as newer ARM designs, particularly the Cortex series, have provided more rigid standards to manufacturers governing memory layout and core components, such as the interrupt controller. This still leaves vendors ample room to customize every aspect of the remaining peripherals, how-

ever. While we focus on the ARM architecture, additional architectures can be added by providing a basic instruction set emulator, creating the short interrupt recording stub, and providing the needed physical memory access to the device to enable recording. Additionally, other architectures use "port-mapped IO" (PMIO) to perform their IO operations. While we do not support this today, PRETENDER could be trivially extended to record these operations instead. All other features of PRETENDER are completely device and architecture-agnostic.

Performance. As PRETENDER involves sending peripheral data and interrupts back and forth between the device and an emulator, this adds some overhead to the firmware's execution. This is particularly noticeable with interrupts, as they tend to be performance- and timing-critical, which could cause issues during recording. This could be overcome through optimization of the implementation, or through the use of purpose-built hardware to interface with the device, as demonstrated in [42].

Obtaining Traces. The principal limitation on the applicability of PRETENDER is not the models or modeling techniques, but in fact the ability to obtain the data to generate them. First, we must be able to obtain a memory data trace for MMIO. In our case study, this is provided via the chip's debug interface, which simply provides access to read and write to any memory address or CPU register. Any interface that also provides this functionality, whether it is an intended debugging interface or one adversarially obtained through an exploit, is sufficient, and could be used to also extract interrupt traces using only this basic requirement. Second, we must be able to observe enough hardware functionality to generate a useful model. This means that we require sufficient code coverage of those code paths that interface with the hardware. We can explore new program behavior using PRETENDER models, but will logically encounter incorrect behavior if these new code paths exercise dramatically different functionality than what

has been recorded. For example, we can re-use our timer model on a completely new firmware that also configures the timer in the same way (e.g., to count up), but not one with a different configuration with vastly different behavior. In our case studies, we utilize human and automated stimulation to achieve maximal coverage during recording, but of course, in the general case, this is an open problem.

Additionally, there are a few aspects of many chips that we simply cannot model correctly with this visibility, particularly Direct Memory Access (DMA) controllers, whose accesses to memory are initiated by the hardware itself, and therefore not visible externally by any conventional means. These are particularly common in higher-speed peripherals, including USB, networking, storage buses, and those common to modern CPUs designed for general-purpose computing. We are unaware of any CPU that allows introspection into DMA activity; however, insight into this problem may be gained by instead observing the firmware's code to locate DMA operations.

Heavily-stateful Peripherals. Not all peripherals, particularly external ones, are well-modeled by a state machine. As we discussed in Section 3.1, we make some assumptions to build a state machine approximation of devices which require it, but this is by no means guaranteed to be correct. One notable case where this will fail is external storage devices, such as SPI-based flash or EEPROM chips. While we could reconstruct much of the traffic to and from these chips seen during recording, reading and writing arbitrary data, as could be possible through a modeled serial port used to provide arbitrary input, will of course not succeed. Fortunately, this problem may be dramatically simplified through high-level modeling, or through the separation of external peripherals from their corresponding internal peripherals, as the behavior of a device as storage may become more apparent.

3.4 Conclusion

In this chapter, we explored the area of firmware re-hosting, and showed that an entirely new class of approaches can enable scalable, thorough program analysis of firmware. As a first step toward achieving this goal, we presented PRETENDER, which generates models of peripherals automatically from recordings of the original hardware. We demonstrated the accuracy and interactivity of these models, by evaluating PRETENDER on multiple firmware samples across different hardware platforms. While there are many open problems remaining before this technique can be generally applicable, we believe this work shows that automated re-hosting is both possible and necessary to ensure that increasingly-important firmware does not go un-analyzed.

Chapter 4

Firmware Re-Hosting through Hardware Abstraction Layer Emulation

The immense diversity of hardware we describe in Chapter 2 significantly complicates the firmware development process for device manufacturers. To mitigate some of the challenges of developing firmware, chip vendors and various third parties provide Hardware Abstraction Layers (HALs). HALs are software libraries that provide high-level hardware operations to the programmer, while hiding details of the particular chip or system on which the firmware executes. This makes porting code between the many similar models from a given vendor, or even between chip vendors, much simpler. Firmware written with HALs are therefore, by design, less tightly coupled to the hardware.

This observation inspired us to design and implement a novel technique to enable scalable emulation of embedded systems through the use of high-level abstraction layers and reusable replacement functionality, known as *High-Level Emulation (HLE)*. Our approach works by first identifying the HAL functions responsible for hardware interactions in a firmware image. Then, it provides simple, analyst-created, high-level replacements, which perform the same conceptual task from the firmware's perspective (*e.g.,* sending an Ethernet packet and acknowledging the action to the firmware).

The first crucial step to enabling high-level emulation is the precise identification

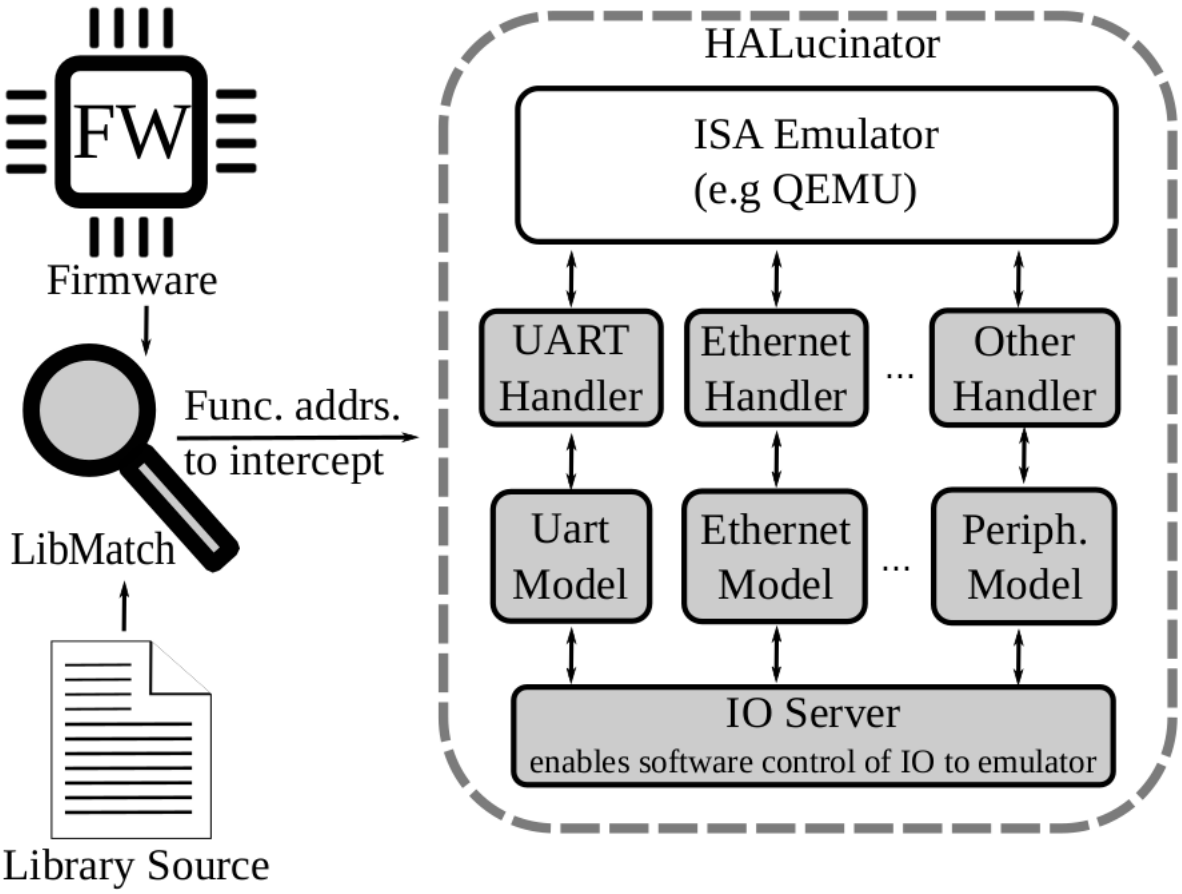

Figure 4.0.1: Overview of HALucinator, with our contribution shown in gray.

of HAL functions within the firmware image. While a developer can re-host their own code by skipping this step, as they have debugging symbols, third-party analysts must untangle library and application code from the stripped binary firmware image. We observe that, to ease development, most HALs are open-source, and are packaged with a particular compiler toolchain in mind. We leverage the availability of source code for HALs to drastically simplify this task.

After HAL function identification, we next substitute our high-level replacements for the HAL functions. While each replacement function (which we term a *handler*) is created manually, this minimal effort scales across chips from the same vendor, and even across firmware using the same middleware libraries. For example, ARM's open-source

mBed OS [38] contains support for over 140 boards and their associated hardware from 16 different manufacturers. By identifying and intercepting the mBed functions in the emulator, we replace the low-level input/output (I/O) interactions—that a generic emulator such as QEMU does *not* support—with high level implementations that provide external interaction, and enable emulation of firmware that uses mBed OS. As an additional effort-saving step, these handlers can make use of *peripheral models*, which serve as the abstraction for generic classes of hardware peripherals (e.g., serial ports, or bus controllers) and serve as the point of interaction between the emulated environment and the host environment, without needing complicated logic of their own. This allows the creation of handlers to also extend across these classes of peripherals, as handlers for any HAL can use the same peripheral models as-is.

Handlers may perform a task as complicated as sending an Ethernet frame through a Direct Memory Access (DMA) peripheral, but their implementation remains straightforward. Most handlers that interact with the outside world merely need to translate the arguments of the HAL function (for example, the Ethernet device to use, a pointer to the data to send, and its length), into the data a peripheral model can use to actually perform a task (e.g., the raw data to be sent). In many cases, the handler does not need to perform any action at all, as some hardware concepts do not even exist in emulation, such as power and clocking.

We assemble these ideas into a prototype system, HALucinator, as shown in Figure 4.0.1, which provides a high-level emulation environment on top of the QEMU emulator. HALucinator supports "blob" firmware, (*i.e.,* a firmware sample in which all code is statically linked into one binary executable) from multiple chip vendors for the ARM Cortex-M architecture. It handles complex peripherals, such as Ethernet, WiFi, and an IEEE 802.15.4 radio (the physical and media access control layers used in ZigBee and 6LoWPAN *i.e.,* IPv6 over Low Power Wireless Personal Area Networks). The

system is capable of emulating the firmware and its interactions with the outside world. We present case studies focused on hybrid emulated environments, wireless networks, and app-enabled devices. HALucinator emulates these systems sufficiently to allow interactive emulation, such that the device can be used for its original intended purpose without its hardware. We additionally show the applicability of HALucinator to security analyses by pairing it with the popular AFL fuzzer, and demonstrate its use in the discovery of security vulnerabilities, without any use of the original hardware. Additionally, the Shellphish CTF team used HALucinator to win the 2019 CSAW Embedded Security Challenge, by leveraging its unique re-hosting, debugging, and fuzzing capabilities [82, 83]. In summary, our contributions in this chapter are as follows:

1. We enable emulation of binary firmware using a generic system emulator (QEMU for us) without relying on the presence of the actual hardware. We achieve this through the novel use of abstraction libraries called HALs, which are already provided by vendors for embedded platforms.

2. We improve upon existing library matching techniques, to better locate functions for interception in the firmware.

3. We present HALucinator, a high-level emulation system capable of interactive emulation and fuzzing firmware through the use of a library of abstract handlers and peripheral models.

4. We show the practicality of our approach through case studies modeled on 16 real-world firmware samples, and demonstrate that HALucinator successfully emulates complex functionality with minimal effort. Through fuzzing the firmware, we find use-after-free, memory disclosure, and exploitable buffer overflow bugs resulting in CVE-2019-9183 and CVE-2019-8359 in Contiki OS [84].

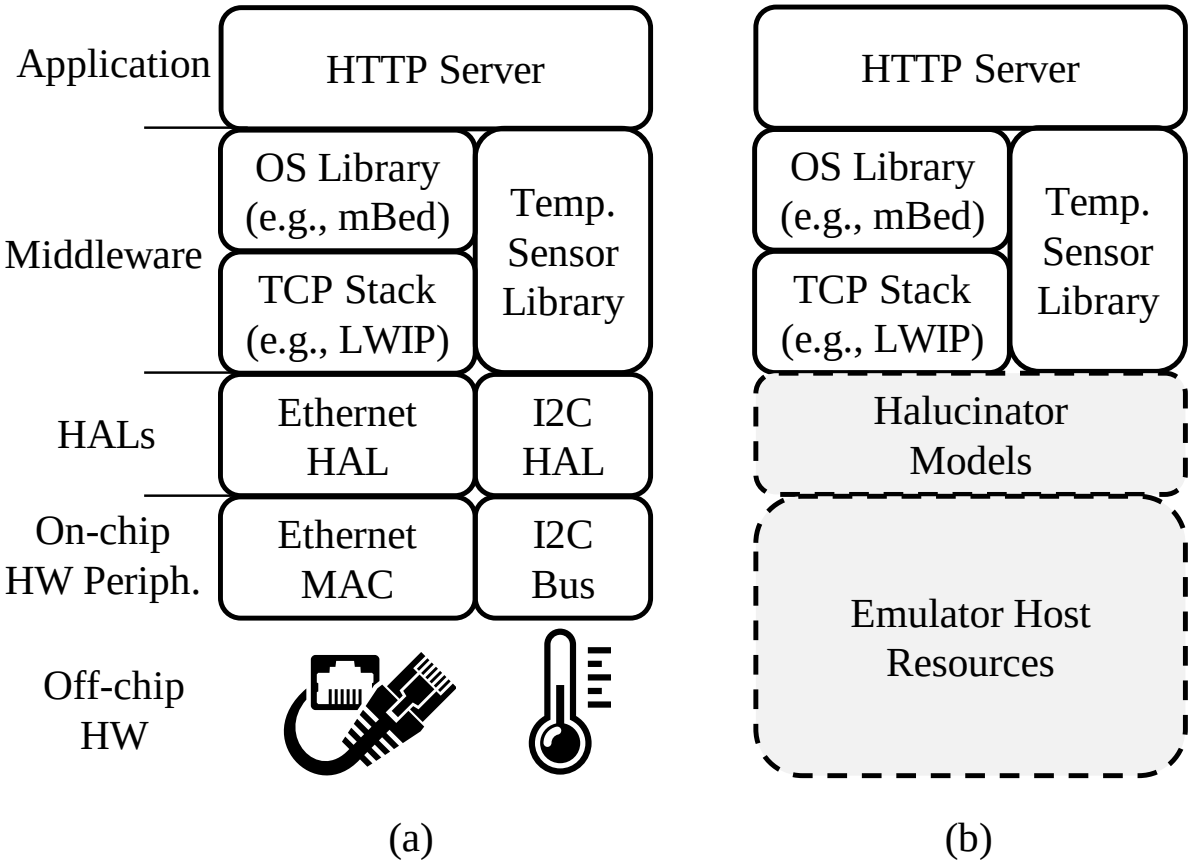

Figure 4.1.2: (a) Software and hardware stack for an illustrative HTTP Server. (b) Conceptual illustration of HTTP Server when executing using HALucinator.

4.1 Motivation

Virtually every complex electronic device has a CPU executing firmware. The increasing complexity of these CPUs and the introduction of ubiquitous connectivity has increased the complexity of firmware. To reduce the burden of creating these devices' firmware, various libraries (*i.e.,* HALs) have been created to abstract away direct hardware interactions.

To make their product portfolios more attractive to developers, microcontroller manufactures are developing HALs and licensing them under permissive terms (*e.g.,* BSD)

to gain a market advantage [85, 86, 87]. HALs provide a common abstraction for families of microcontrollers, thus a single HAL covers many different microcontrollers. For example, STMicroelectronic's STM32Cube HAL covers all their Cortex-M based microcontrollers. As evidence of the investment put into HALs, consider that NXP acquired Freescale in 2015 and currently provides the MCUExpresso HAL—a unified HAL that covers their Cortex-M microcontrollers. Many of these microcontrollers were originally designed by separate companies. It is unlikely NXP would have invested into unifying these HALs if availability of easy to use HALs was not a priority to developers. In addition, the manufacturer's HALs are integrated in their own IDEs [88, 89, 90, 91] and third party development tools (*e.g.,* Keil, IAR). These same HALs are included in embedded OSes (*e.g.,* in FreeRTOS [92], mBed OS [93], RIOT OS [39], and Arduino [94]). These OSes are currently used in commercially available devices [95]. We believe that market pressures to reduce time to market will increase the adoption of HAL's. While we cannot automatically measure the population of devices using HALs today without a large dataset of microcontroller firmware (which does unfortunately not exist), given all of this information, we expect HALs to become ubiquitous in firmware going forward.

Understanding how firmware is built using these HALs is foundational to how HALucinator enables emulation of these firmware samples. Figure 4.1.2a depicts the software and hardware components used in a representative embedded system that HALucinator is designed to emulate. When emulating the system, the on-chip peripherals and off-chip hardware are not present, yet much of the system functionality depends on interactions with these components. For example, in section 4.4 we find that QEMU halts when accessing unsupported (and therefore unmapped) peripherals. The result is all 16 test cases execute less than 39 basic blocks halting on hardware setup, typically clocks, at power up.

4.1.1 The Firmware Stack

The software and hardware stack for an illustrative HTTP server is shown in Figure 4.1.2a. Consider an example where the HTTP server provides the temperature via a webpage. The application gets the temperature using an API from the library provided by the temperature sensor's manufacturer, which in turn uses the I2C HAL provided by the microcontroller manufacturer, to communicate with the off-chip temperature sensor over the I2C bus. When the page containing the temperature is requested, the HTTP server uses the OS library's API to send and receive TCP messages. The OS, in turn, uses a TCP stack provided via another library, *e.g.,* Lightweight IP (lwIP) [96]. lwIP translates the TCP messages to Ethernet frames and uses the Ethernet HAL to send the frames using the physical Ethernet port.

While this is an illustrative example, the complexity of modern devices and pressure to reduce development time is increasingly making it so that functionality in firmware is built on top of a collection of middleware libraries and HALs. Many of these libraries are available from chip manufacturers in their software development kits (SDKs) to attract developers to use their hardware. These SDKs incorporate example applications and middleware libraries including: OS libraries (*e.g.,* mBed OS [38], FreeRTOS [97], and Contiki [84]), protocol stacks (*e.g.,* TCP/IP, 6LoWPAN, and Bluetooth), file systems, and HALs for on-chip peripherals. Each of these libraries abstracts lower-level functionality, decoupling the application from its physical hardware. In order for HALucinator to break the coupling between firmware and hardware, it must intercept one of these layers, middleware/library or HAL, and interpose its replacement functionality instead, as shown in Figure 4.1.2b. Which layer we choose, however, provides trade-offs in terms of generality and reusability of the high-level function replacements, the amount of actual code that we can execute and test, as well as the likelihood of finding a given library in

a target device's firmware. While it is more likely that the author of a given firmware is using the chip vendor's HAL, this bottom-most layer has the largest number of functions, which often have very specific semantics, and often have complex interactions with hardware features, such as interrupts and DMA. At a higher level, such as the network stack or middleware, we may not be able to predict which libraries are in use, but handlers built around these layers can be simpler, and more portable between devices. The chosen layer can also affect the efficacy of some analyses, as we demonstrate in section 4.4. In short, the right answer depends largely on the analyst's goals, and what libraries the firmware uses. In this work, we focus primarily on re-hosting at the HAL level, but also explore high-level emulation approaches targeting other layers, such as the middleware, in our evaluation of HALucinator.

4.1.2 High-Level Emulation

Before discussing the design of HALucinator, we first highlight the ways in which high-level emulation enables scalable emulation of firmware.

First, our approach reduces the emulation effort—instead of manual effort that increases with the number of unique devices, emulation effort increases much more slowly with the number of HALs or middleware libraries, depending on the level where we interpose the function calls. Large groups of devices, from the same manufacturer or device family, share the same programmer-facing library abstractions. For example, STMicroelectronics provides a unified HAL interface for all its Cortex-M devices [85]. Similar higher-level libraries, such as mBed, provide abstractions for devices from multiple manufacturers, and commonly used protocol stacks (*e.g.,* lwIP) abstract details of communication protocols. Intercepting these libraries enables emulating devices from many different manufacturers.

Since HALs abstract away hardware from the programmer, our handlers inherit this simplicity as well. High-level emulation removes the requirement of understanding low-level details of the hardware. Thus, handlers do not need to implement low-level MMIO manipulations, but simply need to intercept the corresponding HAL function, pass desired parameters on to an appropriate peripheral model and return a value that the firmware expects.

Finally, our approach allows flexibility in the fidelity of handlers that we have to develop. For peripherals that the analyst is not concerned with, or which are not necessary in the emulator, simple low-fidelity handlers that bypass the function and return a value indicating successful execution can be used. In cases where external input and output is needed, higher-fidelity handlers enabling communication with the host environment are needed. For example, the function `HAL_TIM_OscConfig` from the STM32Cube HAL configures and calibrates various timer and clock parameters; if not handled, the firmware will enter an infinite loop inside this function. As the emulator has no concept of a configurable clock or oscillator, this function's handler merely needs to return zero, to indicate it executed successfully. On the other hand, a higher-fidelity handler for the `HAL_Ethernet_RX_Frame` and `HAL_Ethernet_TX_Frame` functions that enables sending and receiving Ethernet frames emulates network functionality. Our approach allows for handlers at multiple fidelity levels to co-exist in the same emulation.

4.2 Design

For our design to capitalize on the advantages of high-level emulation, we need to (1) locate the HAL library functions in the firmware (*e.g.,* via library matching), (2) provide high-level replacements for HAL functions, and (3) enable external interaction with the emulated firmware.

HALucinator employs a modular design to facilitate its use with a variety of firmware and analysis situations, as seen in Figure 4.0.1. To introduce the various phases and components of HALucinator, let us consider a simple example firmware which uses a serial port to echo characters sent from an attached computer. Aside from hardware initialization code, this firmware needs only the ability to send and receive serial data. The analyst notices the CPU of the device is an STM32F4 microcontroller, and uses the LibMatch analysis presented in subsection 4.2.2, with a database built for STMicroelectrics' HAL libraries for this chip series. This identifies `HAL_UART_Receive` and `HAL_UART_Transmit` in the binary. The analyst then creates a configuration for HALucinator, indicating that a set of *handlers* (*i.e.,* the high-level function replacements), for the included HAL, should be used. If the handlers do not already exist, the analyst creates them. These two HAL functions take as arguments a reference to a serial port, buffer pointer, and a length. To save effort, these handlers simply translete these arguments to and from a form usable by the *peripheral model* for a serial port (e.g., the raw data to be sent or received). Finally, the *I/O Server* transfers the data between the serial port peripheral model and host machine's terminal. Now, when the firmware executes in HALucinator, the firmware is usable through a terminal like any other console program. This represents only a small fraction of the capabilities of HALucinator, which we will explore in detail in the following sections.

4.2.1 Prerequisites

While HALucinator offers a significant amount of flexibility, there are a few requirements and assumptions regarding the target firmware. First, the analyst must obtain the complete firmware for the device. HALucinator focuses on OS-less "blob" firmware images typically found in microcontrollers. While no hardware is needed during emu-

lation with HALucinator, some details about the original device are needed to know what exactly to emulate. HALucinator requires the basic parameters needed to load the firmware into any emulator, such as architecture, and generic memory layout (e.g., where the Flash and RAM reside within memory).

We assume the analyst can also obtain the libraries, such as HALs, OS library, middleware, or networking stacks they want to emulate, and the toolchain typically used by that chip vendor to compile them. Most chip vendors provide a development environment, or at least a prescribed compiler and basic set of libraries, to avoid complications from customers using a variety of different compiler versions. As such, the set of possible HAL and compiler combinations is assumed to be somewhat small. While firmware developers are free to use whatever toolchain they wish, we expect that the conveniences provided by these libraries and toolchains, and the potential for support from the chip vendor, has convinced a significant number of developers to take advantage of the vendor's toolchain. In section 4.5, we discuss the possibility of using high-level emulation, even in firmware without an automatically identifiable HAL.

HALucinator naturally requires an underlying emulator able to faithfully execute the firmware's code, and able to support HALucinator's instrumentation. This includes a configurable memory layout, the ability to "hook" a specific address in the code to trigger a high-level handler, and the ability to access the emulator's registers and memory to perform the handler's function.

While this may appear to be a long list of requirements, in practice, obtaining them is straightforward. For the ARM Cortex-M devices that we focus on in this work, the general memory map is standardized and available readily from the vendor-provided manual, the location of the firmware in memory can be read from the firmware blob itself, and common emulators such as QEMU [98] faithfully emulate instructions. Each Cortex-M vendor provides open-source HAL(s) for their chips, with compilers and configurations [85, 87,

86, 99]. All that is needed for HALucinator to be applied to a particular device is to obtain the firmware, know the CPU's vendor, and obtain their SDK.

4.2.2 LibMatch

A critical component of high-level emulation is the ability to locate an abstraction in the program which can be used as the basis for emulation. While those developers who wish to re-host their own code, or those interested in open-source firmware projects, can already obtain this information during compilation, analysis of closed-source binary firmware by third parties requires the ability to locate these libraries before emulation can proceed. Existing approaches that address the problem of finding functions in stripped binaries [63, 59, 64] lack support for embedded CPU architectures, particularly the ARM Cortex-M architecture commonly used in many consumer devices and used in this work. While much work has also been done in comparing two binary programs [55, 56], these schemes are not applicable out-of-the-box for comparing a binary with its component libraries.

The nature of firmware itself further complicates library matching. Firmware library functions are typically optimized for size, and two functions with nearly identical code can serve dramatically different purposes. Many smaller HAL functions may simply be a series of preprocessor definitions resolved at compile-time relating to I/O operations, which of course serve different purposes depending on the peripheral being used. One unusual feature of firmware library functions is that they often call functions in the non-library part of the code. With desktop libraries, it is typically expected that library functions are monolithic, *i.e.,* they execute, perform their task, and return to the caller. This is often not true in firmware; common patterns found in HALs include *overrides*, where the developer overrides a weak symbol in the HAL during compilation, or explicit

callbacks, where code pointers are passed in as function arguments. Therefore in order to provide fully-working handlers, we must not only recover the library functions' names and addresses, but those of the application code they call as well.

To address these problems, we create *LibMatch*, which leverages the context of functions within a program to aid in binary-to-library matching. LibMatch creates a database of HAL functions to match by extracting the control-flow graph of the unlinked binary object files of the libraries, plus an Intermediate Representation (IR) of their code. It then performs the following steps to successively refine possible matches:

1: Statistical comparison. We compare three basic metrics—number of basic blocks, CFG edges, and function calls—for each pair of function in the target program and library functions in the database. If functions differ on these three metrics, they are unlikely to be a match, and removing these non-matches early provides a significant performance improvement.

2: Basic Block Comparison. For those pairs of functions that match based on the previous step, we further compare the content of their basic blocks, in terms of an intermediate representation. We consider two functions a match if each of their basic blocks' IR content matches exactly. We do, however, discard known pointers and relative offsets used as pointers, and relocation targets, as these will differ between the library and the binary's IR code. Additionally, unresolvable jump and call targets, even when they are resolvable in the library but not in the binary, are ignored.

While our comparison metric is somewhat naive (i.e., some environmental changes such as compiler, compiler flags, or source code may cause missing matches), and many more complex matching schemes exist (as noted in section 2.3), we make the trade-off that any match is a true, high-confidence match. This trade-off is necessary, as inaccuracies in these direct matches could have cascading effects when used to derrive other matches via

context. Even in the ideal scenario of matching against the exact compiler and library versions, collisions are still expected to occur, as we show in section 4.4.

3: Contextual Matching. The previous step will produce a set of matches, but also a set of collisions, those functions that could not be distinguished from others. We therefore leverage the function's context within the target program to disambiguate these cases, by locating places in the program with matches to infer what other functions could be. While many program diffing tools [55, 56] use two programs' call graphs to refine their matching, we cannot, as our 'second program', is a database of libraries. The libraries in the database are entirely un-linked and have no call graph. We cannot even infer the call graph of a function within a particular library, as HALs may contain many identically-named functions chosen via link-time options. Therefore, we use both *caller context* and *callee context*, to effectively approximate the real call graph of the library functions, disambiguate collisions, and try to provide names for functions that may differ between the library database and the target (*e.g.,* names overridden by the application code, or names outside the libraries entirely).

We first leverage *caller context* to resolve collisions. For each of the possible collided matches, we use the libraries' debugging information to extract the set of called function names. We obtain the same set of called function names from the ambiguous function in the target binary, by using the exact matches for each of the called functions. If the sets of function names in the target and the collided match are identical, the match continues to be valid, and others are discarded. For *callee context*, we gather the set of functions called by any function we were able to match exactly in step two, and name them based on the debug symbols in the library objects. If the function is a collision, it can then be resolved. If the function is not in the database, such as due to overrides by the application, it can then be named. Both of these processes occur recursively, as resolving conflicts in one function may lead to additional matches.

The Final Match. A valid match is identified if a unique name is assigned to a given function in the target binary.

4.2.3 High-level Emulation

After function identification, the emulator must replace the execution of selected functions to ensure the re-hosted firmware executes correctly. These intercepted functions relate to the on-chip or off-chip peripherals of the device, and are implemented manually. To simplify implementation, our design breaks the needed implementation per library into *handlers*, which encode each HAL function's semantics, and *peripheral models* which reflect aspects common to a peripheral type. Under this scenario, each peripheral model only has to be written once, requiring only a small specialized handler for each matched HAL function.

Handlers. We refer to high-level replacements for the HAL's code within the firmware as *handlers*. Creating handlers is done manually, but only needs to be done once for each HAL or library, and is independent of the firmware being analyzed. Each HAL function, even those with the same purpose, will likely vary in terms of function arguments, return value, and exact internal semantics. However, as we will show in section 4.4, almost all handlers are simple, falling into a few basic categories, such as performing trivial actions on a peripheral model, returning a constant value, or doing nothing at all.

Some HALs can be quite large, but most firmware samples only utilize a small fraction of the available functions. In this case, the analyst can follow an iterative process to build handlers. First, the analyst runs the binary in HALucinator, which will report all I/O accesses that are not currently replaced by a handler, and where they occurred. If the firmware gets stuck, or is missing desired behavior, the analyst can evaluate which functions contain the I/O operations, and consider implementing a handler. The process

repeats, and successive handlers produce greater coverage and more accurate functionality. This process can even be performed when the results of library matching are unavailable, or is missing function names required for emulation.

Peripheral Models. Peripheral models intend to handle common intrinsic aspects of what a certain class or type of peripheral must do. They contain little actual logic, but play an important role in creating a common interface between the emulator and the outside world. For example, the peripheral model for a serial port simply has data buffers for transmission and reception of data. When a HAL's serial transmit and receive functions are called, the associated handler can use the peripheral model to trivially perform most, if not all, of its duties in an abstract way.

I/O Server. In order for the re-hosted firmware to meaningfully execute, it must interact with external devices located outside of the CPU. Therefore, in addition to exchanging data with the firmware, each peripheral model also defines an interface for the host system to send data, receive data, and trigger interrupts. These interfaces are then exposed through an I/O server. The I/O server uses a publish/subscribe design pattern, to which peripheral models publish and/or subscribe to specific *topics* that they handle. For example, an Ethernet model will send and receive messages on the '`Ethernet.Frame`' topic, enabling it to connect with other devices that can receive Ethernet frames.

Using the I/O server centralizes external communication with the emulated system, by facilitating multiple use cases without changing the emulator's configuration. For example, the Ethernet model can be connected to: the host Ethernet interface, other emulated systems, or both, by appropriately routing the messages published by the I/O server. In addition, centralizing all I/O enables a program to coordinate all external interactions of an emulated firmware. For example, this program could coordinate pushing buttons, sending/receiving Ethernet frames, and monitoring LED status lights. This

enables powerful multiple interface instrumentation completely in software, and enables dynamic analysis to explore complex internal states of the firmware.

Peripheral Accesses Outside a HAL. Replacing the HAL with handlers and peripheral models simplifies emulating firmware, but occasionally, direct MMIO accesses from the firmware will still occur. These can happen when a developer deliberately breaks the HAL's abstraction and interacts with hardware directly, or when the compiler inlines a HAL function. HALucinator will report all I/O outside handlers to the user. Additionally, all read operations to these areas will return zero, and all writes will be ignored, allowing code that naively interacts with this hardware directly to execute without crashing. We find many MMIO operations, particularly write operations setting peripheral flags and configurations, can be safely ignored as the emulator configures its resources independent of the firmware. We discuss more severe cases, such as firmware not using a HAL, in section 4.5.

4.2.4 Fuzzing with HALucinator

The use of high-level emulation enables the firmware to be used interactively, and also explored through automated dynamic analyses, such as fuzzing. However, fuzzing—especially coverage-guided fuzzing through, e.g., AFL [25]—has different constraints than interactive emulation:

Fuzzed Input. First, the analyst needs to decide how the mutated input should be provided to the target. HALucinator provides a special `fuzz` peripheral model, which when used in a handler, will dispense data from the fuzzer's input stream to the handler. Embedded systems may have multiple sources of input, and this flexibility allows the analyst to chose one or more of them to fuzz.

Termination. Beyond providing input from the fuzzer, the fuzzed firmware must termi-

nate. Current fuzzers generally target desktop programs, and expect them to terminate when input is exhausted; however, firmware never terminates. Thus, we design the `fuzz` model to gracefully exit the program, sending a signal to the fuzzer that the program did not crash during that execution.

Non-determinism. Firmware has significant non-deterministic behavior, which must be removed to allow the fuzzer to gather coverage metrics correctly. This is typically removed from programs via instrumentation, and HALucinator's high-level emulation enables this as well. HALucinator provides static handlers for randomness-producing functions when they are identified, such as `rand()`, `time()`, or vendor-specific functions providing these functionalities.

Timers. One special case of non-determinism are timers, which often appear in microcontrollers as special peripherals that trigger interrupts and other events at a specified interval. Because we cannot guarantee any clock rate for our execution, implementing timers based on real time would lead to non-deterministic behavior, as these timer events can occur at any point in the program. We provide a `Timer` peripheral model, which ties the timer's rate to the number of executed blocks, creating deterministic timer behavior, and fair execution of the timer's interrupt handlers and the main program, regardless of emulation speed.

Crash Detection. Crash detection in embedded systems remains a challenge [9]. A system based on high-level emulation gains a significant amount of crash detection capability from the visibility provided by the emulator, making many generated faults much less silent. Just as with desktop programs, we can instrument firmware to add additional checks. High-level emulation handlers can perform their own checks, such as checking pre-conditions of their arguments (*e.g.,* pointer validity, or positive buffer lengths). High-level emulation can also be used to easily add instrumentation usually handled at

compile-time. For example, HALucinator provides a heap-checking implementation similar to ASAN [100], if the `malloc` and `free` symbols are available.

Input Generation. Finally, fuzzing requires representative inputs to seed its mutation algorithms. HALucinator's fully-interactive mode can be used to interact with the device and log the return values of library calls of interest, which can be used to seed fuzzing. This removes the need for any hardware, even while generating test inputs.

4.3 Implementation

We implement the concept of high-level emulation by creating prototypes of LibMatch and HALucinator targeting the widely-used and highly-diverse Cortex-M microcontrollers.

LibMatch Implementation. LibMatch uses the `angr` [101] binary analysis platform. More specifically, it uses `angr's` VEX-based IR, control-flow graph recovery, and flexible architecture support enables function labeling without any dependence on specific program types or architecture features. Statistics needed for matching are gathered using `angr's` CFG recovery analysis. This includes the basic block content comparisons, which operate on top of the VEX IR statements and their content. Implementing LibMatch for the Cortex-M architecture required extending `angr`. We added support for Cortex-M's calling conventions, missing instructions, function start detection and indirect jump resolution to `angr`. After these extensions, `angr` was able to recover the CFG. When run, LibMatch uses unlinked object files with symbols, obtained by compiling the HAL and middleware libraries to create a database of known functions. It then uses this database to locate functions inside a firmware without symbols. When LibMatch is then run against a firmware sample, it outputs a list of identified functions and their addresses, and makes note of collisions, in the event that a human analyst wishes to resolve them

manually.

HALucinator Implementation. HALucinator is implemented in Python, and uses *Avatar²* to set up a full-system QEMU emulation target and instrument its execution. HALucinator takes as inputs: the memory layout (*i.e.,* size and location of Flash and RAM), a list of functions to intercept with their associated handlers, and the list of functions and addresses from LibMatch. It uses the addresses of the functions to place a breakpoint on the first instruction of each function to be intercepted, and registers the handler to execute when the breakpoint is hit. Note that, while *Avatar²* is typically deployed as a hardware-in-the-loop orchestration scheme, we use it here exclusively for its flexible control of QEMU, and not for any hardware-related purpose.

Handlers are implemented as Python classes, with each function covering one or more functions in the firmware's HAL or libraries. The handlers can read and write the emulator's registers or memory, call functions in the firmware itself, and interact with the peripheral models. Examples of simple and more complex handlers can be found in [102] and [103].

Peripheral models are implemented as Python classes, and can make full use of system libraries or the I/O server to implement the desired functionalities. For example, calls to get the time from a hardware real-time clock can simply invoke the host system's `time()` function. Most models, however, merely act as a store or queue of events, such as queuing received data for the serial port or Ethernet interface.

The I/O server is implemented as a publish-subscribe system using the ZeroMQ [104] messaging library. In addition to serving events to peripheral models from the host system, the I/O server can also connect emulators' peripheral models together, allowing the emulation of multiple interconnected systems. This is particularly useful when the host system has no concept of the interface being shared, such as in the 6LoWPAN examples in section 4.4.

Fuzzing with HALucinator. We created the ability to fuzz firmware using HALucinator by replacing the full-system QEMU engine at the center of HALucinator with AFL-Unicorn [105]. AFL-Unicorn combines the ISA emulation features of QEMU with a flexible API, and provides the coverage instrumentation and fork-server capabilities used by AFL. It lacks any peripheral hardware support, making it unable to fuzz firmware. Adding HALucinator's high-level emulation provides the needed peripheral hardware support. Unicorn and AFL-Unicorn also deliberately remove the concept of interrupts, which are necessary for emulating firmware. Thus, we add a generalized interrupt controller model, that supports ARM's Cortex-M interrupt semantics.

AFL-Unicorn detects crashes by translating various execution errors (e.g., invalid memory accesses, invalid instructions, etc.) into the equivalent process signal fired upon the fuzzed process (e.g., SIGSEGV), providing the appropriate signals to AFL. Models and handlers can also explicitly send these signals to AFL if their assumptions are violated.

4.4 Evaluation

For HALucinator to meet its goal of enabling scalable emulation, it must accurately identify HAL functions in firmware, and enable replacement of those functions with handlers. In addition, the handlers must be created with reasonable effort, and the emulation must be accurate to enable meaningful dynamic analysis of the firmware. In this section, we show that HALucinator meets these goals by evaluating LibMatch's ability to identify HALs in binaries, demonstrating interactive emulation of 16 applications, and then utilizing HALucinator to fuzz network-connected applications.

In our experiments, we use 16 firmware samples provided with different development boards (STM32F479I-Eval [106], STM32-Nucleo F401RE [107], SAM R21 Xplained

Mfg.	Application	HAL Syms	LibMatch w/o Context				LibMatch w/ Context				
			Correct	Coll.	Incorrect	Miss.	Correct	Coll.	Incorrect	Miss.	External
Atmel	SD FatFS	107	76 (71.0%)	22	0	9	98 (91.6%)	2	0	7	3
Atmel	lwIP HTTP	160	128 (80.0%)	20	0	12	144 (90.0%)	9	0	7	8
Atmel	UART	28	24 (85.7%)	2	0	2	26 (92.7%)	1	0	1	1
Atmel	6LoWPAN RX	299	224 (74.9%)	63	2	10	273 (91.3%)	17	4	5	24
Atmel	6LoWPAN TX	300	225 (75.0%)	63	2	10	275 (91.7%)	17	4	4	25
STM	UART	33	15 (45.5%)	17	1	1	23 (69.7%)	9	1	4	6
STM	UDP Echo RX	235	188 (80.0%)	43	0	4	207 (88.1%)	24	0	0	6
STM	UDP Echo TX	235	186 (79.1%)	43	0	4	205 (87.2%)	24	0	0	8
STM	TCP Echo RX	239	192 (80.3%)	43	0	4	211 (88.3%)	24	0	0	5
STM	TCP Echo TX	237	190 (80.2%)	43	0	4	209 (88.2%)	24	0	4	8
STM	SD FatFS	160	111 (69.4%)	47	0	2	140 (87.5%)	20	0	8	5
STM	PLC	495	358 (72.3%))	126	0	11	407 (82.2%)	79	1	8	36
NXP	UART	35	21 (60.0%)	13	0	1	21 (60.0%)	13	0	1	8
NXP	UDP Echo RX	170	133 (78.2%)	25	0	12	141 (83.0%)	16	8	5	22
NXP	TCP Echo RX	176	133 (75.5%)	26	0	17	142(80.7%)	16	8	10	20
NXP	HTTP Server	177	133 (75.1%)	26	0	18	145(82.0%)	16	6	6	20

Table 4.1: LibMatch performance, with and without contextual matching.

Pro [108], NXP FRDM-K64F [109]) from Atmel, NXP, and STM. These samples were chosen for their diverse and complex hardware interactions, including serial communication, file systems on SD cards, Ethernet, 6LoWPAN, and WiFi. They also contain a range of sophisticated application logic, including wireless messaging over 6LoWPAN, a Ladder Logic interpreter, and an HTTP Server with a Common Gateway Interface (CGI). The set of included libraries is also diverse, featuring STMicroelectronics' STM32-Cube HAL [85], NXP's MCUXpresso [86], Atmel's Advanced Software Framework (ASF) [87], lwIP [96], FatFS [110], and Contiki-OS [84], a commonly used OS for low-power wireless sensors, with its networking stack μIP .

Experiment Setup. All STMicroelectronics firmware was compiled using `gcc -Os` targeting a Cortex-M3. The STMicroelectronics boards use Cortex-M4 microcontrollers, however QEMU lacks support for some Cortex-M4 instructions (resulting in a runtime fault), thus these examples were compiled using the Cortex-M3 instruction set. Atmel's example applications were compiled using Atmel Studio 7, using its release build configuration that uses the -Os optimization level and targets the Cortex-M0 ISA (a strict

subset of the Cortex-M3 ISA) as intended for their target board. All NXP samples were compiled using the SDK's "release" configuration, save for using the Cortex-M3 platform instead of M4. All symbols were stripped from the binaries.

4.4.1 Library Identification in Binaries

We first explore the effectiveness of LibMatch in recovering the addresses of functions in a binary firmware program. As there are multiple locations within a firmware that may be hooked, with various trade-offs in the complexity of emulation, here we try to match the entire set of functions provided by the HAL and its associated middleware. We use symbol information in each target firmware sample to provide the ground-truth address of each function. LibMatch then tries to determine the address of each function in its HAL database using a stripped version of this binary.

A comparison of the 16 firmware samples using LibMatch with and without context matching is shown in Table 4.1. LibMatch without context matching is comparable to what is achievable with current matching algorithms (*e.g.,* BinDiff [55], or Diaphora [56]). However, a direct comparison is not possible because these tools only perform a linked-binary to linked-binary comparison and LibMatch must match a linked binary to a collection of unlinked library objects obtained from the HALs and middleware.

In Table 4.1, the number of HAL symbols is the number of library functions present in the firmware, while the 'Correct' column shows the number of those functions correctly identified. The 'Collision', 'Incorrect', and 'Missing' columns delineate reasons LibMatch was unable to correctly identify the unmatched functions. The last column, 'External' is the number of functions external to the HAL libraries that LibMatch with context matching labels correctly. Overall, LibMatch without context matching averaged over the 16 applications matches 74.5% of the library functions, and LibMatch with con-

text matching increases this to an average of 87.4%. Thus, nearly all of the HAL and middleware libraries are accurately located within the binary.

Context matching identifies many of the functions needed for re-hosting firmware. The most dramatic example of this is STMicroelectronics's PLC application; it includes STMicroelectronic's WiFi library, which communicates with the application using a series of callbacks called via overridden symbols. In order to re-host this binary, the handlers for this library must fulfill its contract with the application, by calling these callbacks. Thus, recovering their names, even when they are not part of the library database, is necessary to enable their use during re-hosting. Resolved collisions include various packet handling, timer, and external interrupt functions of the Atmel 6LoWPAN stack, as well as functions needed to enable fuzzing, such as lwIP's IP checksum calculation. One other important category of functions resolved via context includes those that are neither part of the vendor's HAL, nor the application code, but come from the compiling system's standard C libraries, such as `malloc`, `free`, and even the location of the program's `main`.

Collisions are the most common causes of unlabeled functions. Other common causes include C++ virtual function call stubs, and functions that have multiple implementations with different names. For example, the STM32 HAL contains functions `HAL_TIM_PWM_Init` and `HAL_TIM_OC_Init`, whose code is entirely identical, but operate on different data, and have insufficient context to distinguish them. Similarly, in many C++-based HAL functions, a stub is used to lookup and call a method on the object itself; identical code for this can exist in many places. Those without actual direct calls cannot be resolved through context. Finally, many unused interrupt handlers contain the same default content (*e.g.,* causing the device to halt) and thus collide. Since they are interrupt handlers, they are never directly called, and thus cannot be resolved via context. It is worth noting that these cases will confuse any library-matching tool, as there is simply no information on which to make a correct decision within the program.

The few "Incorrect" matches made by LibMatch stem from cases where the library function name actually changed during linking. In these cases, LibMatch has a single match for the function—thus finding a match—but applies the wrong name (i.e., the name before it was changed during linking). Our measure of correctness is the name, and therefore these are marked as "Incorrect". There are two main causes of 'Missing' functions: the application overrides a symbol and we are unable to infer it as an External match via context, or bugs in the CFG recovery performed by `angr` causing the functions' content to differ between the program and the library when they should not. For example, most Cortex-M applications contain a symbol `SystemInit`, which performs hardware specific initialization. Most HALs provide a default, but this symbol is very often overridden by the firmware to configure hardware timing parameters, and it is only ever called from other application-customized code. Thus we lack context to resolve it. None of the unmatched or collided functions are functions needed to perform high-level emulation, and thus, the less-than-100% accuracy of LibMatch does not impact HALucinator.

4.4.2 Scaling of High-Level Emulation

We will examine the benefits of HLE by exploring how the simplicity of handlers and peripheral models allow emulation with a minimum of human effort, and allow this effort to scale to multiple systems.

Handlers and Human Effort. Implementing handlers is a manual task; therefore it is important to quantify the amount of effort required to emulate a system. While we could perform this evaluation in terms of time, or in terms of an objective measure of code complexity (which is given in subsection 4.4.4), these measures do not factor in the amount the analyst actually must understand about the code being replaced, and thus do not fully convey the effort required. Therefore, we divided the handlers used in our

experiments into three categories: *Trivial* handlers simply return a constant—usually indicating the function executed correctly—and require no knowledge of the implementation of the function being intercepted. They are commonly used for hardware initialization functions. *Translating* handlers translate the intercepted function parameters to an action on a peripheral model. They do not implement any logic, but just call a model after getting the appropriate data for the model. This requires knowledge of the function parameters, reading values to be passed to the model, and then writing back values from the model to the appropriate function parameters. For example, the handler for the `ENET_SendFrame` from NXP's HAL, simply reads the frame buffer and length from the function parameters, and passes them to the Ethernet model. The final category, *Internal Logic* is the most complex for HALucinator and requires understanding the internal logic of the replaced functions.

Table 4.2 was created by taking the union of the handlers executed during interactive emulation for the binaries in Table 4.3 and classifying them as trivial, translating, or internal logic. It shows 44.5% are trivial handlers, 42.2% are translating handlers, and 13.3% implement internal logic. Therefore, for our firmware samples, over 85% of the handlers can be implemented with little or no understanding of how the internals of functions they are intercepting are implemented.

The 13% that required understanding internal logic primarily represent cases where the HAL itself manipulated global state also used by the rest of the program. For example, the Atmel Ethernet and 6LowPAN case studies use the external interrupt controller (EXTI) which maps several external interrupts to a single CPU interrupt. The EXTI interrupt service routine (ISR) looks up the ID of the actual interrupt source in an MMIO register, and uses it to look up the correct callback in a global array. HALucinator does not have access to the global array, and thus cannon directly look up the correct callback. Instead, the EXTI handler implements a simple MMIO peripheral that enables

HAL	Trivial	Translation	Internal Logic	Total
ASF v3	12 (30.8%)	19 (48.7%)	8 (20.5%)	39
STM32	17 (58.6%)	9 (31.0%)	3 (10.3%)	29
NXP	8 (53.3%)	7 (46.7%)	0 (0.0%)	15
Total	37 (44.5%)	35 (42.2%)	11 (13.3%)	83

Table 4.2: Categorization by difficulty of implementing handlers. Showing number of handlers that implement Trivial, Translating, and Internal Logic behaviors.

reading/writing the MMIO status register. This enables the EXTI ISR to execute correctly. While this requires understanding some chip-level details, it retains the scaling and relative simplicity of high-level emulation. We implemented a MMIO register and no internal machine, versus implementing *all* the MMIO registers of *all* the used peripherals in the firmware and their associated internal state machines that control how the bits in those registers are used.

Scaling Across Devices. To demonstrate how HLE allows the emulation of one HAL *to scale across devices*, we constructed an experiment using samples from the NXP MCUXpresso HAL, each from a different board and CPU. These represent chips from each of NXPs major ARM microcontroller product families, including Kinetis, LPC, and i.MX, whose designs and peripheral layouts are entirely different due to their development under formerly-separate companies. Regardless of family and lineage, all of these parts share the same HAL. As a result, we obtained 20 instances of the `uart_polling` example, from 20 different development boards. The `uart_polling` example was selected as UARTs are available on nearly every board and the presence of other peripherals varies from board to board. We then emulated these 20 firmware samples using the same NXP UART handlers and peripheral models. Specifically we used three handlers, a transmit handler, receive handler, and a default handler that returns zero. The only differences in the configuration of HALucinator for the different firmware was in the RAM/Flash layout, clock interception, and power initialization functions all of which were handled by the trivial default handler. In total 29 unique functions were intercepted. Six function

				QEMU		Avatar^2			HALucinator			
Mfr.	Application	Software Libraries	Modeled Interfaces	BB	EBC	BB	Fwd R/W	EBC	BB	Funcs.	MMIO	EBC
Atmel	UART	ASF	UART	8	✗	184	467	✗	43	5	4	✓
Atmel	SD FatFs	ASF, FatFS,	UART, SD Card, EXTI	8	✗	344	554	✗	920	14	28	✓
Atmel	lwIP HTTP	ASF, HTTP, lwIP	UART, Ethernet	8	✗	265	935	✗	1,584	8	24	✓
Atmel	6LoWPAN TX	ASF, Contiki	UART, 802.15.4, EXTI	14	✗	121	521	✗	2,734	21	36	✓
Atmel	6LoWPAN RX	ASF, Contiki	UART, 802.15.4, EXTI	14	✗	122	903	✗	2,474	21	36	✓
STM	UART	STM32Cube	UART, GPIO	8	✗	40	17	✗	66	10	7	✓
STM	SD FatFs	STM32Cube FatFS	GPIO, SD Card, Clock	8	✗	41	17	✗	625	18	25	✓
STM	UDP Echo TX	STM32Cube, lwIP	Ethernet, GPIO, EXTI	8	✗	32	15	✗	732	16	10	✓
STM	UDP Echo RX	STM32Cube, lwIP	Ethernet, Clock	8	✗	40	17	✗	568	15	10	✓
STM	TCP Echo TX	STM32Cube, lwIP	Ethernet, Clock, GPIO	8	✗	31	15	✗	1,110	16	10	✓
STM	TCP Echo RX	STM32Cube, lwIP	Ethernet	8	✗	33	15	✗	1,002	15	10	✓
STM	PLC	STM32Cube, STM-WiFi	Timer, WiFI, UART, SPI	39	✗	54	17	✗	713	17	41	✓
NXP	UART	MCUExpresso	UART	4	✗	107	1,766	✓	82	6	28	✓
NXP	UDP Echo RX	MCUExpresso, lwIP	UART, Ethernet	4	✗	54	66	✗	805	13	43	✓
NXP	TCP Echo RX	MCUExpresso, lwIP	UART, Ethernet	4	✗	54	66	✗	1,173	14	43	✓
NXP	HTTP Server	MCUExpresso, lwIP	UART, Ethernet	4	✗	56	68	✗	1,756	14	45	✓
			Averages	**9.7**		**98.7**	**341.2**		**1024.2**	**13.9**	**25.0**	

Table 4.3: Comparison of QEMU, *Avatar^2*, and HALucinator.

at minimum, nine maximum, and 6.9 on average were intercepted per board. This shows that the same handlers and models can be used to support multiple product families. The only challenge was to identify the names of the intercepted clock and power initialization functions.

4.4.3 Interactive Emulation Comparison

Next we re-host the 16 firmware samples shown in Table 4.1 interactively, using QEMU, *Avatar^2* [12], and HALucinator. In this experiment, we use the QEMU provided with *Avatar^2* in its default configuration and load and execute the firmware into QEMU without the hardware present. In this configuration any access to unsupported MMIO in QEMU will fault. *Avatar^2* was configured to execute the firmware in QEMU and forward all MMIO to a physical board connected by a debugger. Thus, all reads and writes to MMIO obtain values from or write to physical hardware. HALucinator utilized the functions found by LibMatch, and we intercept a sufficient number of HAL functions to enable the firmware samples to perform their externally observable func-

tionality as compared to execution on the physical hardware. For any MMIO that is executed, we implement a default MMIO handler that returns zero for reads and silently ignore writes.

We consider the external behavior to be "correct" if *equivalent functionality* can be performed on the emulated system as on the real hardware. Specifically, the TCP/UDP examples successfully transmit the same data as the physical hardware. We are able to access the same pages on the HTTP server firmware samples. The FatFs examples are able to read and write the required data to the the appropriate files within its file system. We verified this by mounting the binary images provide by HALucinator through the SD card model as a FAT32 file system. The 6LoWPAN examples successfully talk to each other and their echoed messages are sent out their UARTs in the same order as the physical hardware. The UART examples are able to send and receive data over their UARTs and give the expected responses. Finally, the PLC sample, connects to its Android programming app, successfully loads a ladder logic, and executes it. Due to the limited inspection capabilities of hardware we cannot verify that equal code paths are followed as compared to physical hardware. Obtaining this level of inspection is a primary motivation for emulating embedded systems. It should be noted that enabling this level of emulation exceeds what is needed purely for fuzzing, as fuzzing can be performed by simply getting the system to read an input. Providing the same level of functionality enables fuzzing to start from a plausible initial starting point, and as will be shown in subsection 4.4.5 HLE enables targeting the fuzzer at different layers within a firmware.

Table 4.3 shows the software libraries used by each firmware, and the interfaces modeled by HALucinator. For each technique it shows the number of unique basic blocks executed ("BB"), which indicates how much of the firmware executes. It also shows if the external input and output behavior matches that observed from executing the firmware

on physical hardware (external behavior correct – "EBC").

For *Avatar²*, we report the number of reads and writes forwarded to the board ("Fwd R/W") which demonstrate that *Avatar²* is correctly forwarding memory requests. For HALucinator, we report the number of functions intercepted ("Funcs") and the number of unique addresses handled by the default MMIO. The number of functions intercepted gives a measure of how much work is required to emulate the firmware using HALucinator, and the MMIO using the default handler are accesses to hardware that could potentially be replaced with further interception of HAL functions.

HALucinator enables the correct black-box behavior in *all cases*—all vendors, all boards, all firmware samples. Among our baseline approaches, the NXP UART firmware using *Avatar²* is the only other firmware successfully emulated. This is because it is a simple firmware that polls the MMIO and does not use any interrupts. In all cases, QEMU triggers a bus fault when any MMIO occurs and executes at most 39 unique basic blocks (on STM PLC). *Avatar²*'s MMIO forwarding enables executing further into the firmware (the average number of basic blocks increases from 9.7 to 98.7), but quickly runs into problems. All the STM samples and the NXP UDP, TCP, and HTTP samples enable the SysTick timer early in their initialization. The SysTick timer is part of the Cortex-M architecture and implemented in QEMU. The emulation is significantly slower than the actual hardware thus, when SysTick is enabled QEMU is quickly overwhelmed with interrupts. It is unable to finish handling one interrupt before the next occurs. HALucinator intercepts the HAL functions that initialize the SysTick timer and substitutes a counter to keep time; enabling it to avoid this problem. All the Atmel firmware samples halt when the debugger fails to write an MMIO address on the board. The debugger does not give any indication why this occurs. In most cases, the debugger has successfully written the address previously, implying the error is not that the address is invalid. This highlights one of the challenges of emulating with hardware-in-the-loop.

The emulator, debugger, and board must be synchronized and execute without error in unison to enable successful emulation. Even if the debugger worked reliably, the firmware samples depend on interrupts, which *Avatar²* does not synchronize with the emulator and thus they would still fail to execute correctly.

This experiment shows how HALucinator enables the emulation of complex firmware that exhibits the same external functionality as the firmware executing on real hardware, which existing approaches cannot do. HALucinator executed more than 1,000 basic blocks on average, 10x more than *Avatar²*, on our sample firmware. The emulation of four different boards from three different manufactures demonstrates the ability of HLE to support a wide variety of hardware, and the reuse of the same peripheral models for all boards shows their scalability across vendors and hardware platforms.

4.4.4 Code Complexity Metrics

To assess the difficulty and complexity of the required manual effort when programming the handlers and peripheral model, we examine the amount of code—in source lines of code (SLOC)—and its cyclomatic complexity (CC) in Table 4.4. Let us look at the largest handler for each peripheral. The ASF Ethernet handler requires 119 SLOC across with an average function cyclomatic complexity of 1.9 and a maximum of 6. The Ethernet peripheral model takes an additional 60 SLOC with average cyclomatic complexity of 2.2. This means an Ethernet interface can be emulated in under 200 lines of simple code.

However, firmware uses more than one peripheral. The 6LoWPAN firmware samples use the IEEE 802.15.4 radio, UART, Clock, the external interrupt controller (EXTI), and on-board debugger (EDBG) interfaces. For these firmware samples the amount of code and complexity of the code is low. It require 228 SLOC for the handlers and 177 SLOC

lines of code for the peripheral models with the highest average cyclomatic complexity being 2.2. Thus, with 405 lines of simple code, we emulate the firmware for a wireless sensor implementing the 6LoWPAN protocol.

4.4.5 Fuzzing with HALucinator

We now demonstrate that HALucinator's emulation is useful for dynamic analysis by fuzzing the network connected firmware shown in Table 4.5, and the firmware used in the experiments in WYCINWYC [9]. WYCINWYC investigates the observability of memory corruption on embedded systems, and provides a vulnerable implementation of an XML parser on embedded system. Experiments were performed on a 12-core/24-thread Xeon server, with 96GB RAM. Table 4.5 shows the statistics provided by AFL during the fuzzing sessions. Crucially, we were able to scale these experiments to the full capacity of this hardware, due to removing the dependence on the original hardware.

We include the WYCNINWYC example here, as it provides a benchmark of crash detection in an embedded environment. This firmware uses the same STM HAL used in previous experiments, and no additional handlers were implemented. We substituted our `fuzz` model for the serial port model, and fuzzing was seeded with the non-crashing XML input included with the binary. We triggered four of the five crashes in [9], without the need for additional crash detection instrumentation, and were able to trigger the final crash by simply adding the ASAN-style sanitizer described in subsection 4.2.4. The remaining firmware were re-hosted as in the interactive experiments, replacing the I/O server with the `fuzz` model for network components and adding fuzzing-related instrumentation. We also provided handlers for disabling library-provided non-deterministic behaviors (e.g., `rand()`), and generated inputs by simply recording valid interactions performed in the previous experiments, and serializing them into a form that can be

mutated by AFL.

These experiments uncovered bugs in the firmware samples. The ST-PLC firmware implements a Programmable Logic controller that executes uploaded ladder logic programs. It uses WiFi connectivity to receive the ladder logic programs from an Android app. This sample is extremely timer-driven, and made use of the deterministic timer mechanism to ensure that each input produced the same block information for AFL. We provided AFL with only a minimal sample ladder logic program obtained from the STM PLC's Android app by capturing network traffic. After only a few minutes, AFL detected an out-of-bounds memory access; upon further inspection, we identified a buffer overflow in the firmware's global data section, which could result in arbitrary code execution. The vulnerability is previously unknown, and we are working with the vendor on a mitigation.

The Atmel HTTP server firmware is a small HTML and AJAX application running on top of the popular lwIP TCP/IP stack. After nearly 9 days, AFL detected 267 "unique" crashes, which we disambiguated to 37 crashes using the included minimization tools. Manual examination revealed the crashes related to two bugs: a heap double-free in lwIP itself, and a heap use-after-free caused by the HTTP server's erroneous use of lwIP functions that perform heap management. The firmware, and the Atmel ASF SDK itself ships with an outdated version of lwIP (version 1.4.1), and both issues have since been fixed by the lwIP developers.

However, random mutations in Ethernet frames, even guided by AFL, are not likely to produce much coverage in the core application logic of the firmware. To focus more directly on the HTTP server, and not the IP stack, we can exploit the flexibility of high-level emulation, and instead re-host the binary in terms of the TCP APIs of the lwIP library (discovered by LibMatch) that the HTTP server itself was written with, allowing the fuzzed packets to reach deeper into the program. Fuzzing at the higher level quickly found a buffer over-read in the HTTP server's handling of GET request parsing, which

provides an information disclosure in the heap.

The three crashes in the 6LoWPAN sample correspond to a buffer overflow in the handling of the reassembly of fragmented packets, resulting in overwriting many objects in the binary's data section with controlled input, and eventually remote code execution. The issue relates to the Contiki-OS platform, and as in the previous example, has been fixed since the version included in the latest SDK was produced. However, the fix in the latest version introduced two critical vulnerabilities, which we reported as CVE-2019-8359 and CVE-2019-9183 respectively. We worked with the Contiki authors to patch these bugs.

These experiments show that HALucinator enables practical security analysis of firmware without massive re-engineering effort and *without* any hardware. The scalability is in both the types of firmware that can be emulated, and the number of instances that can be concurrently emulated. This enables large parallelization of analyses and testing such as fuzzing. The discovery of bugs in real firmware samples demonstrates that the emulation is useful for dynamic analysis of complex firmware.

4.4.6 Evaluation of P2IM Firmware Samples

In order to test the applicability of HALucinator to realistic firmware, we obtained a portion of the case-study samples used in the real-world firmware evaluation of P2IM [35]. These samples represent multiple CPU manufacturers, and various HAL implementations, as described in Table 6 of the paper.

We obtained and re-hosted the five samples from this set that take input from outside the device. For the PLC, Heat Press, and car controller, the firmware contained the Arduino platform HAL, and we implemented handlers for a small subset of the Arduino platform's functions, comprising only five new handlers, to allow these samples to run.

As this HAL is designed for those new to embedded programming, it helpfully abstracts away all hardware-specific features, making it a natural fit for our technique. As a result, this meant that all handlers fell into the *Trivial* or *Translating* categories. The drone firmware contains the STM32 HAL used extensively in our evaluation in subsection 4.4.3; we added three additional *Translating* handlers, and the firmware ran without issue.

Finally, the console uses RIOT OS [39], which is both an RTOS kernel and a set of hardware abstractions and drivers. RIOT OS exposes a standard set of functions for hardware peripherals, with multiple implementations depending on the chip in use. Of the seven new handlers that were required, five fell into the *Trivial* or *Translating* categories. However, there was one notable exception: the RIOT task switcher uses new ARM architectural features and CPU instructions not yet supported by QEMU or Unicorn Engine. Thankfully, this is a standard component of RIOT that, like any other, can be turned into a handler. By implementing the context switching as a handler (requiring 15 lines of handler code), we both get deep introspection into the behavior of RIOT OS programs, and the ability to explore multi-threading-related issues in RIOT OS programs in the future, regardless of their underlying hardware.

We fuzzed these samples with HALucinator. Table 4.6 shows the results. We observed a variance in execution speed, both due to the nature and size of the input, but also how well this input is checked for correctness. For example, the Drone sample executed particularly slowly, due to the fact that if erroneous input was detected, the firmware would call an error handler routine, which caused the system to hang. We were able to reproduce the crashes in the PLC and Heat Poress samples.

4.5 Limitations and Discussion

We believe that LibMatch and HALucinator represent an important step in the practicality and scalability of the dynamic analysis of embedded firmware. However, the problem in general is not fully solved. Here we will discuss limitations, and open problems in embedded firmware analysis.

Use and Availability of HALs. The process of high-level emulation as described in this work, requires the firmware use a HAL, and the HAL must be available to the analyst (e.g., either open source, or part of the microcontroller's SDK). The compilation environment for the LibMatch database must be similar to the compilation environment for the firmware, and QEMU must support the microcontroller architecture. Even when these conditions are met, handlers and peripheral models must be developed for each HAL. Progress on any of these limitations will increase the applicability of HALucinator in analyzing firmware.

We note that microcontroller vendors are investing significant resources into the development of HALs and license them under permissive terms. While we cannot estimate the population of devices today that use HALs, we expect these steps on the part of manufacturers will lead to a rapid increase in HAL usage. However, if a HAL is not used in a firmware sample, or is unavailable to the analyst, then LibMatch cannot be used for identifying interfaces usable for high-level emulation. This does not prohibit high-level emulation; as a reverse-engineer could manually identify useful abstractions in the binary. Which would still be preferable to writing low-level QEMU peripherals.

Library Matching. LibMatch implements extensions on top of library matching algorithms that allow them to be used for the purpose of finding HALs and libraries in firmware. However, we note that the effectiveness of LibMatch, especially when the compiler or library versions used is unknown, is limited. This limitation comes from

function matching techniques' inability to cope with compiler-induced variations in generated code. While partial techniques have been proposed, most recently in [64], the problem is not solved in the general case. High-level emulation and LibMatch will benefit directly from any advancement in this orthogonal problem area of function matching in the future. LibMatch's primary contribution is the use of context (callees/callers) of a function to disambiguate binary equivalent functions, which is necessary to enable correct interception and replacement of functions by HALucinator.

4.6 Conclusion

In this chapter, we explored the concept of high-level emulation to aid in the practical re-hosting and analysis of embedded "blob" firmware. To find useful abstractions, we showcased improvements in binary library matching to enable hardware abstraction layers and other common libraries to be detected in binary firmware images. Implementations were then broken down into abstract components that are reusable across firmware samples and chip models.

HALucinator is the first system to combine these techniques into a system for both interactive dynamic analysis, as well as fuzzing. We re-hosted 16 firmware samples, across CPUs and HALs from three different vendors, and with a variety of complex peripherals. High-level emulation made this process simple, allowing for re-hosting to take place with little human effort, and no invasive access to the real hardware. Finally, we demonstrated HALucinator's applications to security, by using it to detect security bugs in firmware samples. We believe that high-level emulation will enable analysts to broadly explore embedded firmware samples for fuzz testing and other analyses. HALucinator is available at `https://github.com/embedded-sec/halucinator`, hal-fuzz is available at `https://github.com/ucsb-seclab/hal-fuzz`.

	STM32 Handlers			Atmel Handlers			NXP Handlers			Peripheral Model		
		CC			CC			CC			CC	
Peripheral	SLOC	Max	Ave	SLOC	Max	Ave	SLOC	Max	Ave	SLOC	Max	Ave
802.15.4		—		89	3	1.4		—		62	3	2.0
Clock	21	1	1.0	25	2	1.3		—			—	
EDBG		—		30	2	1.6		—			—	
Ethernet	67	4	1.5	119	6	1.9	50	2	1.2	60	3	2.2
EXTI		—		47	4	2.2		—		32	2	1.4
GPIO	46	1	1.0		—			—		36	2	1.3
SD Card	82	5	1.7	116	3	1.5		—		60	4	2.3
SPI	55	1	1.0		—			—		66	5	1.9
WiFi TCP	69	8	2.4		—			—		59	5	2.2
Timers	77	1	1.0	61	2	1.3		—		43	2	1.7
UART	29	1	1.0	37	1	1.0	36	1	1.0	41	4	2.0

Table 4.4: Showing SLOC, maximum and average cyclomatic complexity (CC) of the handlers written for the STM32, Atmel, and NXP HALs and the associated peripheral models.

Name	Time	Executions	Total Paths	Crashes
WYCINWYC	1d:0h	1,548,582	612	5
Atmel lwIP HTTP (Ethernet)	19d:4h	37,948,954	8,081	273
Atmel lwIP HTTP (TCP)	0d:10h	2,645,393	1,090	38
Atmel 6LoWPAN TX	1d:10	1,876,531	23,982	0
Atmel 6LoWPAN RX	1d:10	2,306,569	38,788	3
STM UDP RX	3d:8h	19,214,779	3,261	0
STM UDP TX	3d:8h	12,703,448	3,794	0
STM TCP RX	3d:8h	16,356,129	4,848	0
STM TCP TX	3d:8h	16,723,950	5,012	0
STM ST-PLC	1d:10h	456,368	772	27
NXP TCP RX	14d:0h	218,214,107	5164	0
NXP UDP RX	14d:0h	240,720,229	3032	0
NXP HTTP Server	14d:0h	186,839,871	9710	0

Table 4.5: Fuzzing experiments results.

Name	Time	Executions	Total Paths	Crashes
PLC	9d1h	167,649,720	1,585	634
Heat Press	9d1h	55,577,331	991	13
Steering Ctlr	23d14h	98,393,268	469	0
Drone	4d1h	9,234,661	4666	0
Console	4d1h	124,442,630	2834	0

Table 4.6: P2IM case-study firmware sample fuzzing results

Chapter 5

Security Retrofitting for Monolithic Embedded Firmware

In this chapter, we will explore the reality of security retrofitting monolithic binary firmware images. We first identify the concrete pre-requisites and challenges an analyst needs to consider to perform retrofitting on a given device. Then, we will propose novel automated reverse-engineering techniques able to guide the analyst through the process to the maximum extent possible. While the immense hardware and software diversity in monolithic firmware-based devices does not allow for a full automation of the retrofitting process, our techniques enable an analyst to produce a patch with only basic knowledge of the device's hardware and the flaw that needs patching. Specifically, our techniques perform three fundamental steps that are needed to retrofit firmware: (1) identifying attacker-controlled sources of input, (2) identifying memory locations suitable for inserting a patch, and (3) identifying verification mechanisms that prevent the deployment of a patch.

We combine the aforementioned components into a system, SHIMWARE, able to perform all of these tasks on a firmware image and insert a patch payload to mitigate a given vulnerability. Our system is based on the popular open-source `angr` [101] binary analysis framework, which allows for minimizing effort while the handling of the diverse

architectures and binary formats found in firmware.

We first evaluate the capabilities of our system to identify the firmware's sources of input on a dataset of both synthetic and real-world firmware images, and show that the system is able to locate the IO-related code of a program with a low false-positive rate. Then, we showcase the effectiveness of our system by retrofitting fixes for severe security- and safety-critical vulnerabilities in three real-world devices: a high-end Programmable Logic Controller (PLC) found in factory and military equipment, a Bluetooth-enabled cardiac implant monitoring device, and a network-enabled laboratory power supply. These devices contain vulnerabilities that are not the result of an implementation error, but a significant defect in the design of the device itself.

In summary, our contributions are as follows:

- We examine and enumerate the challenges inherent in the security retrofitting of real-world embedded devices, and highlight why current approaches are incompatible with monolithic firmware images.
- We propose novel analyses able to automatically reverse-engineer a firmware image, and provide an analyst with the information needed to locate sources of attacker-controlled input, safely insert a patch payload, and ensure that self-checks preventing such modifications are bypassed.
- We assemble these techniques into a system, SHIMWARE, and show its generality and effectiveness both through synthetic evaluation, and through the mitigation of severe logic vulnerabilities in three real-world, safety-critical devices.

In the spirit of open research, we will release all our analyses and code as open-source upon publication.

5.1 Challenges & Goals

In this section, we walk through the process that an analyst uses to create a patch for an embedded system, and discuss the challenges inherent in each step and our solutions to these problems. At first glance, the process would seem to entail simply taking the firmware code, altering some bytes in it to fix the vulnerability, and running the new version; however, the reality of embedded systems—and even the best-practices advocated by the security community—can hinder third-party security retrofitting.

Obtaining the Firmware Code. For desktop programs, this step is trivial; the analyst already has the program, as they are able to run it. In an embedded system, such as one based on monolithic firmware, this is not trivial. Unfortunately, vendors making their firmware available is incredibly rare, necessitating the extraction of firmware: either from the device itself, or from a mobile or desktop application designed to update it.

Many device vendors consider their firmware to be protected intellectual property. To help address this concern in their products, CPU vendors have come up with numerous hardware mechanisms to prevent the firmware inside their SoCs from being extracted. Manufacturers commonly include hardware debugging features such as JTAG in their devices to ease the hardware development life cycle, but best-practices dictate disabling this in production devices as it can be used to read the firmware. Some vendors have begun implementing forms of protection in the flash memory itself to prevent reading the firmware, even when JTAG is enabled [111, 112]. Vendors also include anti-tamper mechanisms [113] which can erase the firmware if an extraction is attempted, or even if the device's case is opened.

To obtain this firmware, therefore, the analyst must obtain the device and examine it to see which protections are enabled; if they are not, they and can use readily available hardware debugging tools to extract the firmware. These hardware mechanisms can also

be vulnerable to their own implementation vulnerabilities [114, 115]. Nevertheless, if the vendor has taken simple steps to prevent their firmware from being read, the device cannot be patched by a third-party.

Creating a Patch. A security patch should, by definition, have an effect related to the processing of input from an attacker-controlled source. With no standard sources of input, but numerous hardware peripherals that can generate input data, the analyst currently has the tedious task of manually reverse-engineering the firmware and hardware to find the location in the firmware where data from the outside world is accessed. Although we know that the location and function of MMIO peripherals and registers will vary, we can assume these locations to be fixed at compile-time, and they can be found in the program as constant pointers to a peripheral or specific register. However, there are numerous places where hardware is accessed in ways that do not constitute input; serial ports and busses are generally more useful for security retrofitting than timers, clocks, and power controls.

Therefore, we propose *IOFinder*, an analysis that uses static analysis and symbolic execution to locate those functions that produce data that affects the rest of the program. This analysis filters out irrelevant data to only show the actual input, which can be used to make security-related decisions in a patch.

Inserting a Patch. Once it has been developed, a patch must be inserted into the firmware image. On a system whose firmware contains a normal filesystem, this could be as simple as replacing an ELF file. As we target monolithic firmware images, we are left with the more difficult challenge of finding a place in the firmware to safely add code, without affecting the original functionality. Unfortunately, we cannot simply insert code next to the source of attacker-controlled data and shift the remainder of the binary, due to the known-hard problem of locating and adjusting all pointers which would become

incorrect due to the shift. Therefore, the most effective option is to insert the additional code in an unused region of the firmware sample, and substitute an instruction near the source of data for a branch to this region [24]. Deciding which regions are safe to use, however, is its own challenge. We develop techniques that help an analyst to find safe locations to place the patch within the firmware image. As a result, our approach, which we call *LocationFinder*, finds either known-unused space on the device's flash memory, or known-expendable code regions.

Deploying a Patch. Most firmware designs include some sort of verification mechanism to verify its integrity, either when the system boots, or when it is upgraded. These mechanisms can be divided into two categories: those designed to prevent accidental modification (such as CRCs and checksums), and those that are designed to stop intentional modification (such as cryptographic signatures). Which checks we must deal with also depends on our firmware injection vector. We typically have the choice of using either the firmware's own update mechanism to deploy our retrofit or using a hardware injection mechanism such as JTAG or direct flash memory access. If we find an unprotected JTAG port on a device and the firmware's digital signatures are only checked during an over-the-air update, we can bypass this entirely by flashing our own firmware via JTAG. Therefore, in order to successfully deploy a patch, we must mitigate any self-check that affects either our chosen firmware installation vector or the firmware's boot process.

We thus propose an analysis, *SelfCheckFinder*, able to identify many forms of self-checks present in firmware by looking for operations utilizing the content of the firmware itself.

In summary, the state of modern firmware protections and hardware countermeasures makes it difficult for an analyst or a tool to patch and deploy a monolithic firmware image. We discuss the conflict between security best-practices and security retrofitting

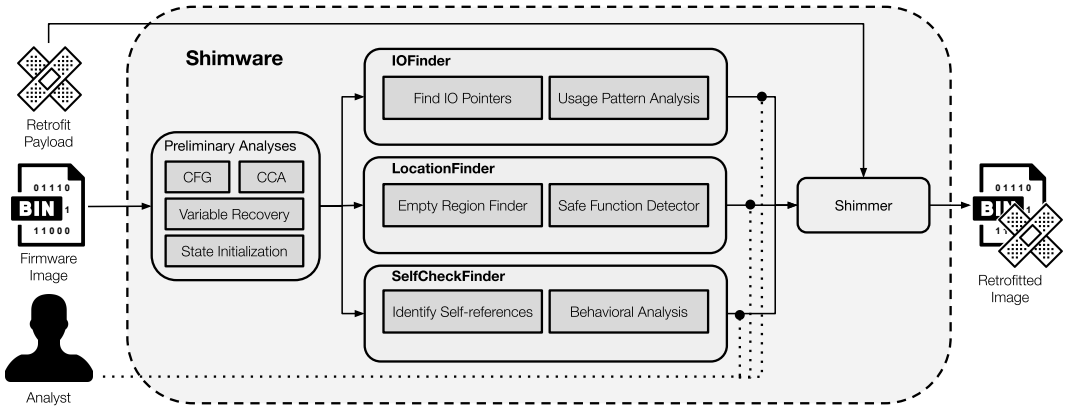

Figure 5.2.1: SHIMWAREoverview. Our static-symbolic analyses automatically identify (a) attacker controlled sources of input, (b) memory regions where to insert a patch payload, and (c) self-checks that prevent firmware modifications. An analysts can then leverage the extracted information to instruct the Shimmer to retrofit the firmware image.

in Section 5.5. In this work, we aim to aid the analyst in retrofitting firmware, where possible, by automating the tasks of finding sources of attacker-controlled data, safe code injection locations, and code self-checks.

5.2 Methodology

In this section, we propose our automated program analysis approach to simplify the process of security retrofitting monolithic embedded firmware. We identify three tasks that represent time-consuming, firmware-specific efforts, and design program analysis techniques to automate them. The outputs of these analyses can be combined to allow an analyst to successfully inject a retrofit payload into a firmware image.

The system combining these techniques, which we call SHIMWARE, performs the following steps (Figure 5.2.1):

IOFinder: The analyst must locate the source of the potential attacker's input within the firmware. This analysis locates usage of MMIO-related pointers in the binary,

and then applies symbolic execution to understand how they are used. This allows us to filter the vast majority of these usages that are uninteresting to the analyst, e.g., peripheral configuration and status checks, and instead focus on those where data from the peripheral enters and affects the rest of the program. The result of this analysis is a set of functions that read or write data from MMIO.

LocationFinder: The LocationFinder analysis guides the analyst in choosing a region where to insert their code. We combine possible injection locations through the identification of known-unused regions, as well as known-expendible functions, and present the analyst with information on how much space is available from each source.

SelfCheckFinder: This analysis locates functions in the binary that check their own content. These can be designed to detect accidental modification of the binary due to hardware glitches (e.g., CRCs and checksums), as well as intentional modification (e.g., digital signatures). We first statically prune the list of all functions to those that could possibly constitute a self-check, and then perform symbolic execution on the remaining set to determine if a self-check is performed. The result of this analysis is a set of functions that contain self-checks, which should either be mitigated (e.g., by adjusting the firmware's checksum) or removed altogether.

Shimmer: The final step is to assemble the retrofitted firmware image, and deploy it onto the device. This phase of the system takes the analyst's retrofit payload, the analysts choices about which IO functions to intercept, where to put the payload, and which self-checks to remove, and creates a ready-to-use firmware image.

5.2.1 Pre-requisites and Assumptions

There are some pre-requisites that must be satisfied in order to use SHIMWAREand its techniques.

Device Access. The device to be retrofitted must be available to the analyst, and some basic details about the system, such as its basic functionality and CPU model, must be known. The analyst can provide SHIMWAREwith the CPU model in order to get accurate information on the hardware peripherals accessed by the firmware. However, the analyst must also know which of the many possible sources of input in the firmware image they wish to safeguard. For example, a firmware image could be communicated with via multiple serial ports, but if one of these is connected to a radio module that can be communicated with by an attacker, this is clearly the interface to be inspected. All of this information can be obtained quickly through examination of the device.

Firmware Access. The firmware must be obtained in full, either from the manufacturer, an associated desktop or smartphone app, or from the device itself. Many devices split their firmware into two parts: a bootloader and a primary application. While the SelfCheckFinder can spot all the needed checks, many of these checks are found within the bootloader, so this code is needed as well. An unprotected hardware debug interface, such as JTAG, provides all of the needed access, but directly interfacing with flash chips or reverse-engineering of the device's update mechanism may also be sufficient [116].

Patch Deployment. The analyst needs a way to deploy a retrofitted firmware image onto the device. An unprotected debug interface, or reverse-engineered update mechanism suffices here, but as discussed in Section 6, some hardware countermeasures may prevent the use of modified firmware images.

Availability of Test Corpora. Determining whether or not a patch does not alter the device's functionality is known to be undecidable in the general case. Therefore, we

assume the analyst has access to sufficient tests suites or companion apps to ensure the proper function of the device after the shimmed firmware is applied.

While this may seem like a long list of requirements, we demonstrate that they are reasonable by applying SHIMWAREto real-world devices in Section 5.4.

5.2.2 IOFinder

The first step to defend a device from attack is to figure out where that attack is possibly coming from. Since we deal with monolithic firmware without function names or library information, and cannot rely on the presence of a standard library that provides IO functionality, we must reverse-engineer the binary to find where the attacker's input comes from.

As we outline in Section 5.1, there are two significant complications with locating IO. First, in most modern architectures, particularly ARM, we cannot tell statically which instructions in the binary perform IO operations, as normal load and store instructions are used to access peripherals. Second, the location, layout, and semantics of each hardware peripheral varies widely with the CPU on which the firmware is designed to run. Even when these accesses are located, firmware images perform numerous IO operations that are of no interest to the analyst, such as setting and clearing configuration flags, or checking status registers. When retrofitting, we are only interested in the input sources that handle actual data.

That said, we are able to leverage a few key insights to make this task tractable through automation. First, and most importantly, the location of MMIO-based peripherals is fixed by the hardware and known at compile-time by the firmware's compiler. These pointers are often stored in global memory, or used as function arguments to the IO-related functions to select which peripheral to use. While this indirection must be

resolved, we know that all IO must result from one of these constant pointers. While the exact semantics of peripherals varies, these peripherals are all located in pre-defined regions of the address space that are fixed by the instruction set architecture standard. Finally, even this semantic information can be resolved to some extent through the use of the same peripheral labeling information available to debugging tools.

We combine these insights into our IOFinder, which uses a hybrid static-symbolic approach to locate interesting IO functions. This analysis performs the following steps.

Compute the Fully-initialized State. A common pattern in firmware is to store global pointers, structs, or objects representing the configured IO devices in global RAM instead of hard-coding them into the program. These are often initialized at the beginning of the firmware's boot, far from where they are actually used. As a result, to know which functions in the program perform MMIO operations, we need to compute the state of the program after these initializations occur. To do this, we created a novel static analysis that locates and performs any assignment of a constant pointer into global memory, and creates a state consisting of the union of all such initializations.

Find IO Pointers. Leveraging the fully-initialized state, we scan the binary for references from the code to the architecturally-defined IO region. We also scan global memory locations previously found to be initialized to an IO pointer. Since any IO activity must include one of these pointers, the result of this step is the set of all functions that contain such an access.

Usage Pattern Analysis. Since all of these pointers are usually not declared or accessed near where the actual IO operation occurs (e.g., they are passed into another function as an argument), we utilize symbolic execution to determine how these pointers are used and locate specific IO operations of interest. We chose every function that defines a pointer to an IO memory area, or uses a global memory location that is initialized to an IO pointer,

as entry points for symbolic execution. Since many IO operations are inconsequential to the analyst, instead of immediately logging all IO operations encountered, we apply dynamic taint tracking to understand how data is used. During the symbolic execution, we use the following rules:

- When data is read from an MMIO peripheral, we taint the resulting data.
- When data is written to an MMIO peripheral, we examine the expression relating to the data to be written. If data being written was previously also read from MMIO, this is likely a *read-modify-write* pattern for setting and clearing flags, and is discarded.
- When data is written to an MMIO peripheral, if the data is constant, it likely was not a result of a meaningful behavior from the rest of the program (e.g., flags and configuration) and is ignored.
- When data is written to an MMIO peripheral, and is not from IO or a constant, it must have come from a function argument or global memory, and is logged as a source of output.
- When data is written to non-MMIO memory, we check if the destination is related to a function argument (e.g., a pointer to a buffer) or global memory. If we are writing to such a location, and the data came from MMIO, this is logged as it represents a location in which data from MMIO is made available to the rest of the program.
- When the starting function returns and data from MMIO is returned by value, this also logged as it represents IO data being made available to the outside program.

- When data read from MMIO is itself used as a pointer, this is logged as being a likely source of Direct Memory Access (DMA), as embedded devices tend to handle pointers to the buffers they are operating on.

This produces a set of CPU instructions that perform some kind of input or output accessible to the surrounding program, along with the exact MMIO address that was targeted.

External Peripheral Information. The analyst must now decide which peripheral is relevant to their retrofitting scenario, so that its data can be used to develop a retrofit payload. Inevitably, firmware can access data from peripherals which are uninteresting to the analyst, in the same manner as those which represent external input. While these are not false-positives in the traditional sense, we can use some information about the microcontroller's hardware to help the analyst quickly locate peripherals related to their retrofitting scenario by labeling the names and registers of peripherals accessed during the symbolic execution step. For example, on the ARM architecture, SystemView Debugger files (SVD) are available in an online repository [49] for many popular embedded ARM CPUs, which can be queried by IOFinder to label the results with names and descriptions.

Listing 5.2.2 shows a set of IO-related functions from the `atmel_6lowpan_udp_rx` firmware binary, taken from the dataset used in Chapter 4. This binary implements a 802.15.4 mesh network node, with a radio module controlled over the Atmel SAMR21's SPI bus. During the firmware's boot, the function `trx_spi_init()` is called, which sets up the SPI bus selected at compile-time to control the radio and stores a pointer to it in a global struct. Much later, when data is received, the firmware's interrupt handler calls `trx_reg_read()`, to obtain data from the radio. If we were to examine `spi_trx_reg_read()` without its names, and without any other context, we would see a function that adds an offset to a value in memory, and dereferences it, which may or may

not be an IO operation, depending on what this global memory value (`&master`) is. When we compute the fully-initialized state, however, we notice `&master->hw` stores a pointer to the SPI bus controller (0x42001800), and this is added to our fully-initialized state. When we run our IOFinder using this state, we notice that `spi_master->SPI.DATA.reg` is a pointer to the IO region and taint the variable `data`. When the value of `data` is returned, the analysis records this as a MMIO read operation, since the data from the SPI bus is being made available to the rest of the program. Since we know that this sample was built for an Atmel ATSAMR21G18A, we can use the available SVD files for this chip to automatically label this access as coming from the `SERCOM4->I2CM_DATA` register, the data register of one of the chip's combined SPI/I2C/USART interfaces.

5.2.3 LocationFinder

To retrofit a monolithic firmware image, we need to find a location in the binary where we can insert the payload. However, as discussed in Section 5.1, this is not simple. Therefore, we have to find a safe region within the binary to insert our payload without disrupting the device's functionality.

While this problem can be shown to be undecidable—demonstrating that a memory region is not used by a program requires solving the halting problem—we can make some assumptions to find regions in the program that are highly unlikely to be used.

First, we consider monolithic firmware images, which come from non-volatile storage in embedded systems. One characteristic of this storage is that it is not easily written to; in order to perform a write operation, a program likely needs to manipulate an MMIO-based peripheral, erase an entire page of the flash, and replace it with a new one. As a result, this means that unlike highly volatile data residing in RAM, and our firmware image can be assumed to be relatively static.

Second, we may not always have sufficient free space to insert our payload; firmware binaries typically need to fit into relatively small storage spaces, and are usually compiled with sized-focused optimizations. Without any guarantees on available insertion space, we are left to remove something from the binary itself to make room. This too can be shown to be undecidable; furthermore, existing work on this area [67, 76] relies on having complete, accurate control-flow graph information, which is not possible in this setting.

To address these issues, we implement the LocationFinder leveraging a series of heuristics to find available regions in the binary.

EmptyRegionFinder. This analysis locates regions of the firmware that are highly likely to be unused. We locate contiguous regions of repeating values (typically 0 or 0xFF), and track the largest one found in the binary. We ignore regions that are statically referenced in the binary, such as when pointers referring to the region are used in the program. While we cannot guarantee that a pointer to a nearby location is not used to access a seemingly-empty region, such as in a loop, we note that it is highly unlikely to find such a region in the firmware, given the access constraints on non-volatile storage, without it being legitimately unused.

SafeFunctionDetector. We locate functions that are safe to remove from the binary. To sidestep the undecidability of this problem, we make a very conservative definition of the functions we wish to remove. In some embedded systems, particularly those with safety-critical roles, functions that test hardware's correct behavior are occasionally present. An interesting property of such functions is that these functions appear meaningless from a purely software-focused perspective. For example, a function that tests memory and registers might perform actions such as writing a pattern, reading it back, and making sure the values before and after are equivalent. We note that in the absence of severe hardware failure, removing such functions does not, by definition, alter

the behavior of the program.

Therefore, we use targeted symbolic execution to identify such functions. We symbolically execute every function in the binary, after statically pruning functions that cannot meet this definition. If a function branches based on its input, calls a function, writes to non-stack variables, or returns a value based on its input, we cannot guarantee it is safe to remove. In other words, if every path constraint in the function simplifies to `true`, and the function is void or returns a constant, we can eliminate it. This narrow definition makes this analysis fast by constraining the amount of execution needed to make a determination.

The LocationFinder uses both of these analyses, and picks the largest available region to inject the analyst's payload. As we mention in Section 5.2.1, we use available testing corpora, such as test programs and companion apps, to verify that these results are correct.

5.2.4 SelfCheckFinder

To deploy a retrofitted firmware image, we have to locate any places where the firmware checks its own integrity so that they can be mitigated or removed.

Defining what a self-check is, however, must be done very carefully. Cryptographic functions are a logical tool for implementing self-checks, and previous work [117, 118] proposes various static and dynamic approaches to finding cryptographic functions. There are plenty of self-checks (e.g., the simple addition-based checksum) that would not be detected by these schemes, but we would still need to locate them here. Moreover, not all cryptographic functions are self-checks; we only are interested in those which actually involve the content of the firmware. Finally, modern SoCs include hardware support for CRCs and cryptography, meaning that a self-check's actual math operations may not

appear in the code at all.

To find self-checks, we make two key observations: First, similar to the IOFinder, our "self-check" must utilize a pointer to the beginning of the region it wishes to check. Theoretically, this is the base address of the binary, but we note that monolithic firmware images that are internally composed of a bootloader and an application may have a scheme in which one region checks another. Therefore, we define a *self-reference* to be a pointer to the firmware, which is evenly divisible by 0x400. This was chosen to encode the intuition that a distinct region in the firmware (or the entire firmware as a whole) is likely aligned to the beginning of a page of its flash memory, which is typically a multiple of 0x400.

Second, we know that, with such a pointer, there is some kind of loop that uses the pointer to access the binary, in order to compute the self-check. As a result, there is likely a single instruction in this loop that reads from a large number of locations within the binary.

With these two ideas in mind, the SelfCheckFinder proceeds as follows.

Compute the Fully-initialized State. Using the same technique described in Section 5.2.2, we compute a fully-initialized state, but this time considering only pointers that are self-references.

Identify Self-references. We prune the list of all functions in the binary to contain only those that use a self-reference, or use global memory known to contain a self-reference.

Behavioral Analysis. We employ symbolic execution on the remaining functions, and look for places where the same instruction reads from many locations in the firmware. Precisely, we consider a function a self-check if it accesses at least N locations, where N is also the number of loop iterations allowed during the execution. In short, if for every new iteration, we get an additional access to the firmware, this loop likely implements a

self-check. We exclude from this set any function whose loop also writes to N locations; these include common primitives for string processing such as `memcpy` or `memmove`.

The analysis produces a list of self-checks, which should be bypassed in order for the firmware to boot when modified. The analyst will determine that this result is correct by actually performing a retrofitting; the firmware will not boot if the self-checks are not removed.

5.3 Implementation

We implemented SHIMWAREin Python using the `angr`[101] binary analysis framework. This allowed us to easily implement both static and dynamic symbolic analyses in the same framework. `angr` also offers broad architecture support, and the flexibility to handle monolithic firmware images.

Loading the Binary. In order to perform analyses on any monolithic firmware image, we first have to obtain the base address, entry point, and architecture of the binary. Doing this automatically is a known open problem, but we leveraged an `angr` plugin that automates this process for many ARM-based firmware images [119], including all of the binaries used in this work. This produces a representation of the device's memory with the loaded firmware.

Static Analyses. Once the binary is loaded into simulated memory, we perform static control-flow graph recovery, which also locates the boundaries of functions using function prologue scanning. We also perform calling convention analysis to determine the number of arguments and return values for each function, as well as locating distinct program variables using Variable Recovery [120].

Finally, we collect cross-references in the binary, which allows us to determine where constant code or data locations are used. These analyses give us the data to implement

the subsequent components, and are provided by pre-existing components of the `angr` framework.

Computation of the fully-initialized state, used in both IOFinder and SelfCheckFinder, is implemented using `angr`'s constant propagation and cross-reference collection analyses. We perform constant propagation for every function in the binary, in topological order, following the program's callgraph. Any value known to be a pointer which is propagated into memory is reflected in the fully-initialized state. For the purposes of this analysis, we define a constant pointer as any integer that could represent an address in the architecturally-defined RAM or IO regions. These constant pointers are also propagated into function calls; a function is only propagated after all of its callers, and will be propagated multiple times, one for each set of arguments used at a callsite. Any conflicting memory writes result in a value of *TOP* being stored in the initialized state. We then collect statically-resolvable cross-references to global memory that had IO addresses stored in them during initialization. We select any function that references the initialized memory locations for further analysis in IOFinder or SelfCheckFinder.

Symbolic Execution. Our analyses use under-constrained symbolic execution to examine a function's behavior, by starting at each function's first instruction, and using the fully-initialized state. To make this tractable on large binaries, we used a number of `angr`'s Exploration Techniques to limit how much execution we perform. We limited the number of basic blocks in any given path to 10,000, the amount of time spent on a function to 5 minutes, as well as using depth-first search and removal of dead-ended paths to reduce memory usage. For IOFinder, we also used the taint tracking and adaptive inter-function level mechanism proposed in previous work [121], to focus our analysis on relevant portions of code.

Shimmer. The final step of SHIMWAREis to assemble all of the information collected

in the previous steps, and the analyst's manually-created payload, into a modified image that can be deployed to the device. While the content of the analyst's payload is outside the scope of this work, Shimmer allows for payloads to be written in C, after which they are compiled to match the target device's sub-architecture, and size-optimized into a binary blob. The Shimmer module implements a model of retrofitting in which the analyst's payload is triggered right after input is read in from the potentially attacker-controlled source found in the IOFinder analysis. First, the analyst must chose which source of input to monitor, from the list found in the IOFinder analysis. Second, they need to chose from the available payload injection locations with enough space to hold the payload. Finally, they can opt to simply automatically remove all the self-checks, or mitigate them manually (e.g., by adjusting the firmware's checksum).

5.4 Evaluation

To assess the capabilities of our tool, we first perform an in-depth evaluation of our IOFinder on a synthetic dataset collected from related work. Then, we present three case studies where we show how SHIMWAREsuccessfully led to security retrofitting of three real-world, safety-critical devices.

5.4.1 IOFinder Evaluation

First, we explore the performance of the IOFinder analysis, on firmware samples obtained from previous work. While SelfCheckFinder and LocationFinder locate features that are only present in production commercial firmware [1], we can use development board and open-source firmware samples, for which source code and symbols are avail-

[1]Note that we did run LocationFinder and SelfCheckFinder on these samples, and they correctly produced no output.

able, to serve as ground truth for the analysis. To build our dataset, we obtained 17 from related work [35] and from the dataset created in Chapter 4. We used all available samples that were built as "bare-metal"; we discuss challenges with mbedOS, Arduino, and other library-OS frameworks in Section 5.5. These samples represent five microcontroller models, from three vendors, with widely-varying peripheral and software driver implementations, and a diverse set of applications, including a PLC, CNC mill, and mesh network nodes. Using these samples, and the hardware for which they were built, we enumerated the set of peripherals that actively communicate with the outside world, as these are the peripherals an analyst would potentially consider as a source of data for a security retrofit. This process was manual, to account for dead code, and to consider only those peripherals which actually transmit and receive data externally. This specifically includes peripherals such as serial ports, busses, and sensors, and excludes timers, clocks, power control, and other common peripherals that do not constitute communication.

Table 5.1 shows our results; the MCU column indicates which microcontroller model the firmware was designed for, and the "Useful Peripherals" column lists the peripherals determined to be useful for retrofitting. The "Tot. No. Functions" column shows the number of functions in the binary, as identified by `angr`'s static analyses, while "No. Candidate Functions" refers to how many of those were selected for further investigation by our static heuristics. During the dynamic phase of our analysis, the symbolic execution generated many load and store operations from the IO region ("Tot. No. IO Ops" column), of which a much smaller number ("No. Filtered IO Ops.") survived our heuristics and are considered potentially useful to the analyst. Finally, the "No. Useful IO Ops." column shows how many of those operations flagged by the analysis were related to the set of useful peripherals.

With regards to our static analysis phase, the results show that we are able to effectively focus our analysis on the part of the program containing IO, while our computation

of the fully-initialized state allows us to draw proper correlations between IO initialization functions and later uses.

During the dynamic phase, the results show that in many cases we are able to significantly reduce the amount of IO the analyst would need to consider. However, we note that the difference between the useful set of peripheral accesses and the set of filtered IO operations reported to the analyst does not constitute false-positives in the conventional sense. All of the reported IO operations are indeed valid IO operations, and are useful for various reverse-engineering tasks. Even peripherals that are not important for the task of shimming, such as clocks and timers, can have their data stored and made available to the rest of the program, and would be reported as such. IOFinder automatically applies external labeling information from the `cmsis-svd` [49] database, which labels each peripheral register for all supported CPUs, allowing an analyst to quickly locate any peripheral of interest in the output. Actual false-positives could theoretically result when the underlying static analyses are unable to determine the correct calling convention of functions (e.g., `angr` determines a function returns a value to the caller, when it does not), although we did not notice any such cases in the output of this experiment. The analysis does, however, have a few false-negatives ("No. Missed IO Ops." column). These cases all stem from an inability to determine a correlation between an IO-related initialization function and an actual IO function, due to the use of nested structs and C++ objects to store the IO configuration. To test this, we implemented a mode for IOFinder where the analyst can manually designate IO-related structs, and we were then able to locate all of the missing peripherals. Future advances in binary type recovery related to structs will help us determine this information automatically.

Table 5.1: IOFinder Evaluation Results. For each of the 17 samples in our dataset, we report the MCU model, the useful peripherals, and the results of our analysis in terms of reported IO functions and operations. Our filtered IO operations contain all the useful ones.

Sample [Dataset]	MCU Model	Useful Peripherals	Tot. No. Functions.	No. Candidate Functions	Tot. No. IO Ops.	No. Filtered IO Ops.	No. Useful IO Ops.	No. Missed IO Ops.	Actual Useful Peripherals
atmel_6lowpan_udp_tx	ATSAMR21G18A	ETH(SPI),I2C,UART	533	182	91	46	21	2†	ETH(SPI),I2C
atmel_6lowpan_udp_rx	ATSAMR21G18A	ETH(SPI),I2C,UART	533	182	91	46	21	2†	ETH(SPI),I2C
p2im_cnc [35]	STM32F429	UART	331	121	110	32	3	0	UART
p2im_drone [35]	STM32F103	UART,I2C	230	71	39	38	18	0	UART,I2C
p2im_robot [35]	STM32F103	I2C,UART	205	41	59	22	15	0	I2C,UART
p2im_soldering_iron [35]	STM32F103	DMA(ADC),I2C(IMU),I2C(OLED)	371	99	107	40	19	4†	DMA(ADC),I2C(IMU)
samr21_http	ATSAMR21G18A	UART,ETH(SPI)	324	68	61	26	8	0	UART,ETH(SPI)
samr21_uart_polling	ATSAMR21G18A	UART	44	35	28	16	3	0	UART
samr21_fatfs_usd	ATSAMR21G18A	SPI(SDIO),UART	207	189	51	25	13	0	SPI(SDIO),UART
st-plc	STM32F401	UART(Wifi),SPI,ADC	981	278	142	49	11	0	UART(Wifi),SPI,ADC
stm32_tcp_echo_client	STM32F469	ETH,I2C	477	63	86	25	19	0	ETH,I2C
stm32_tcp_echo_server	STM32F469	ETH,I2C	478	62	80	26	19	0	ETH,I2C
stm32_udp_echo_client	STM32F469	ETH,I2C	468	58	86	25	19	0	ETH,I2C
stm32_udp_echo_server	STM32F469	ETH,I2C	463	58	80	25	19	0	ETH,I2C
nxp_uart_polling	MK64F12	UART	108	33	36	11	4	0	UART
stm32_fatfs_usd	STM32F469	I2C,SDIO	276	42	81	33	26	0	I2C,SDIO
nxp_fatfs_usd	MK64F12	SDHC,UART	240	59	64	17	6	0	SDHC,UART

†: We managed to cover these false negatives by providing our system with additional knowledge about the employed data structures (Section 5.4.1).

5.4.2 Case Study: Power Supply

We used SHIMWAREto retrofit an RD DPS5015 [122] lab power supply unit (Figure 5.4.3). This unit allows an engineer in a scientific or industrial setting to adjust the voltage and current available on the device's front panel connectors, to power and test devices during development, or for lab experiments. However, like many modern lab power supplies, it also has communications capabilities, to allow for remote automation, over RS485, Bluetooth, or WiFi, depending on the configuration. The unit contains an STMicro STM32F100 ARM Cortex-M3-based CPU, and accesses all remote communications mechanisms over a serial port.

Unfortunately, the legitimate functionality of this communication mechanism can allow anyone with network or radio proximity to the device to remove all safety limits, and adjust the voltage or current to any value, causing damage or destruction of the

device-under-test, and potentially of the unit itself [123]. While simply disabling network connectivity is one way to make this device safe, we would instead like to add the functionality that the voltage limits specified by the operator on the physical device represent a maximum of what the remote automation can set.

We obtained the popular OpenDPS firmware used with this device [124] by installing it onto our unit, and then dumping it via the device's exposed SWD debugging port. This yielded a monolithic firmware image, in which `angr`'s CFG recovery detected 420 functions. The IOFinder analysis detected functions containing IO references, out of which 109 actually performed interesting IO when executed. Among these was a clearly-labeled access to the USART1 data register (USART1->DR), which would serve as a source of input data. LocationFinder discovered a region of 872 bytes between what appears to be the bootloader and the primary application of the firmware. The SelfCheckFinder located exactly one self-check: the use of CRC16 to validate the firmware image.

We were successfully able to retrofit our voltage-limiting inside the firmware, and deployed it over the SWD debugging port. To test the correct functionality of the device, we used the device's front panel to adjust the voltage and current settings, and manually triggered each of the device's configuration menu items. We also tried the remote communication features, and verified that the only difference was that we were unable to set the maximum voltage higher than the one set on the front panel.

5.4.3 Case Study: PLC

We employed SHIMWAREto retrofit an Allen-Bradley ControlLogix 1756 PLC [125]. This high-end, but end-of-life, product is suitable for large factory automation tasks, and comes in the form of a chassis with at least one CPU card, and numerous IO devices depending on the application. Our configuration contains the L64 CPU card, an

analog output card, and an Ethernet card. The CPU contains a custom ARM7TDMI-S derivative CPU, with numerous custom ASIC components controlling network and communications features, presenting a unique challenge for our approach. While we do not know the MMIO layout of the main CPU, we know we are looking for input from the chassis backplane, a proprietary high-speed bus, which is therefore likely to be using some form of DMA.

Unfortunately, as with many PLCs, this unit suffers from a similar issue as the power supply mentioned above: anyone on the same network can control it completely by default. While the device can be configured to not accept any network traffic (e.g., via the front keyswitch) this dramatically diminishes the usefulness of the device in a modern automation setting. This device is also well past its end of support, and will no longer receive updates from the manufacturer. To this end, we wish to add built-in safeguards to not allow an attacker to adjust the parameters of a running ladder-logic program outside of safe parameters.

We obtained the firmware through the manufacturer's website. CFG recovery yielded 8,937 functions in the binary. Of these, 593 had IO pointers. When executed, IOFinder detected 340 unique IO operations. Since the CPU is custom, we had to perform the additional step of reverse-engineering the CPU's MMIO layout, but were helped by our analyses in doing so, as we could focus our efforts on those portions of the code detected by IOFinder. Since the amount of detected IO locations is numerous, and we estimate that this result is legitimate, we focused on those locations which performed DMA-based operations. This only consisted of 31 IO-related instructions, representing 5 unique MMIO registers. All of these appeared to be related to a peripheral at 0x40000000, which turned out to be the DMA controller for the backplane.

SelfCheckFinder detected 2 self-checks, the checksum and CRC, calculated on the whole firmware image, which can either be trivially removed, or manually adjusted to

match after modification.

The LocationFinder's results for this device were unusual, there were very few empty regions, all of a small size. This is because this firmware explicitly checked many of its empty regions to ensure that they were indeed empty, therefore generating memory accesses that led our analysis to discarded such regions. However, LocationFinder was able to locate a very large (5.7k) function, which it could prove was safe to remove, giving us ample room for a payload. This function appears to implement a test of the system's ALU and registers, such as performing math, and storing and recalling the results. From a mathematical perspective, this extremely large function, which seems to consist of large, unrolled, loops, entirely simplifies away when executed symbolically. The function also returns no value, instead calling an assert-fail function if an error occurs. In our testing, simply removing this function was sufficient, and produced no noticeable change in the program's behavior.

Using the above information provided by our analyses, we injected a payload into the firmware, which is able to filter incoming Common Industrial Protocol (CIP) messages, to ensure that no message alters the variables of a running ladder-logic program outside of compiled-in parameters. We tested the system for correct behavior by running a test ladder-logic program on the device, and by connecting the RSLogix companion software to monitor the device's behavior. Our ladder logic program causes an LED connected to one of the PLC's output cards to blink at a certain rate (see Figure 5.4.4); we used our payload to clamp this values between 2 and 10 Hz. We verified that, if a CIP message is received that would set this rate outside the bounds, that it is ignored, and a message is sent to the operator. We should also note that, like many safety-critical devices, this PLC has numerous self-tests (other than the one we removed), watchdog timers, and hardware safeguards, both at boot-time and continuously at run-time, which would have alerted us to any faults due to our retrofit.

5.4.4 Case Study: Pacemaker Monitor

We used SHIMWAREwith a *[Redacted for responsible disclosure]* pacemaker monitor. This small hand-held device acts as a bridge between the patient's phone (via Bluetooth) and a pacemaker (via short-range radio). The patient is instructed by the device's companion app to use the device daily to transmit cardiac data to their doctor.

Unfortunately, we discovered a flaw in how the device employs cryptography to keep unauthorized apps and devices from connecting to it. This allows any attacker in range of the device to connect to it, and issue a broader set of commands to the pacemaker than the manufacturer intended, along with accessing the contained medical data. For ethical reasons, we will not speculate on any effects this may have on the implant itself. We reported this flaw to the vendor, and are working with them on an official fix. Nevertheless, the root cause here is one we can fix via a retrofit, namely stopping the original cryptographic bruteforce attack. The device contains an STM32F103 Cortex-M3 CPU, and accesses its Bluetooth controller via a serial port. An unprotected JTAG port is present, via a pogo pin connector.

While we were able to obtain the firmware through the companion Android app, we dumped the full firmware, including its bootloader, from the JTAG port. CFG recovery yielded 3,687 total functions. IOFinder yielded 696 candidate IO functions. During dynamic analysis, this produced 93 potentially-useful IO operations, which were automatically labeled with their peripheral names. Among these was the USART3 peripheral operating in DMA mode, our desired source of IO. LocationFinder found a large (250k) unreferenced region of flash memory at 0x80bf96c, between the firmware itself and the non-volatile storage portion of the flash memory. SelfCheckFinder located 5 self-checks, 3 of which were false-positives, and two of which were hardware-backed CRC32 of the entire firmware, performed at boot-time. These false-positives were related to the firmware

upgrade routine itself, which naturally loops over the firmware and manipulates the flash memory controller. That said, if we were to hypothetically remove all detected self-checks, the firmware would work, excluding the update mechanism, which may be advisable to remove anyway when using retrofitted firmware.

We tested the LocationFinder and SelfCheckFinder's results by inserting our shim into the firmware, and deploying it over JTAG. When the serial port connected to the Bluetooth chip is written to, we check the outgoing payload against the error message sent when an incorrect cryptographic key is used. If this message is found, we reset the device, causing a significant delay, and dramatically lowering the chance of a successful attack. Because this device is only intended to be powered on for a few minutes at a time, this will sufficiently mitigate the risks, until a manufacturer-provided patch which fully mitigates the underlying design issue is available. The device's companion app has exactly one function: downloading the implant's data and sending it to a doctor. It has no menus or configuration, and after the various steps (connecting via Bluetooth, connecting to the implant, and downloading the data), will display a large green checkmark, indicating that it succeeded. We ensured that this process still worked as expected on our retrofitted unit, and functionality was not impacted.

5.5 Limitations

While we showed SHIMWAREin action on diverse, real-world devices, our approach has a few limitations.

IOFinder. Our IOFinder currently does not work effectively on firmware created on top of large firmware frameworks, such as Arduino [126] or ARM's mbed [38], which utilize a high level of abstraction, and object-oriented programming techniques to simplify development, and ease porting to many hardware platforms. As these requirements do not

exist in final commercial devices, vendors instead opt to use lighter-weight packages, such as those provided by the compiler vendor, or creating their own. A careful reader can note that these characteristics seem to be the opposite of a library matching approach ([127] and Chapter 4), as the additional, standardized code in these packages is simpler to locate by matching. Therefore these two approaches are directly complimentary; library matching works well on larger libraries where static analysis fails, but IOFinder works in situations where the libraries to match cannot be obtained, or are customized.

Both the IOFinder and LocationFinder rely on `angr`'s calling convention analysis to determine whether a function returns data. As this depends on the completeness of the CFG to work, SHIMWAREcould hypothetically produce false alerts due to the inability to determine the proper calling convention of a function. That said, we did not notice any such alerts during our experiments.

5.6 Conclusion

In this chapter, we explored the challenges and solutions to security retrofitting of monolithic embedded firmware. We identified the tasks of locating the attacker controlled IO, finding safe payload injection locations, and mitigating self-check functions as candidates for automation. To this end, we proposed three novel static-symbolic analyses that locate these features in a firmware image automatically, and significantly reduce the analyst effort. Our prototype system, SHIMWARE, combines these techniques with a tool to inject an analyst-developed payload into the firmware. In addition to testing the IOFinder analysis on firmware samples whose IO locations are known, we employed the full system to address three safety and security-critical vulnerabilities in off-the-shelf products from the engineering, healthcare, and industrial automation sectors. Our results show the promise of security retrofitting, even in the challenging context of monolithic

firmware, and we hope that future advances in securing embedded firmware will continue to allow users and analysts alike to secure their devices, when manufacturers may not.

```
// SPI bus connects to the 802.15.4 radio
void trx_spi_init() {
  ...
  // master is a global, 0x42001800 is the SPI controller
  // Both are propagated into spi_init
  spi_init(&master, (Sercom *const)0x42001800, &config);
  ...
}
status_code __fastcall spi_init(spi_module *const module, Sercom *const hw, ...) {
  ...
  // The address 0x42001800 is stored into &master
  module->hw = hw;
  ...
}
// ... much later ...
uint8_t __fastcall trx_reg_read(uint8_t addr) {
  ...
  spi_master = master.hw;
  data = spi_master->SPI.DATA.reg & 0x1FF; // variable data is tainted
  ...
  return data; // MMIO_READ detected via return value
}
```

Figure 5.2.2: Example code from the `atmel_6lowpan_udp_rx` firmware from Chapter 4. Without resolving indirection statically, we cannot see the pointer to the SPI bus being used in `trx_reg_read()`. Names of functions and variables provided only for clarity and not known during analysis.

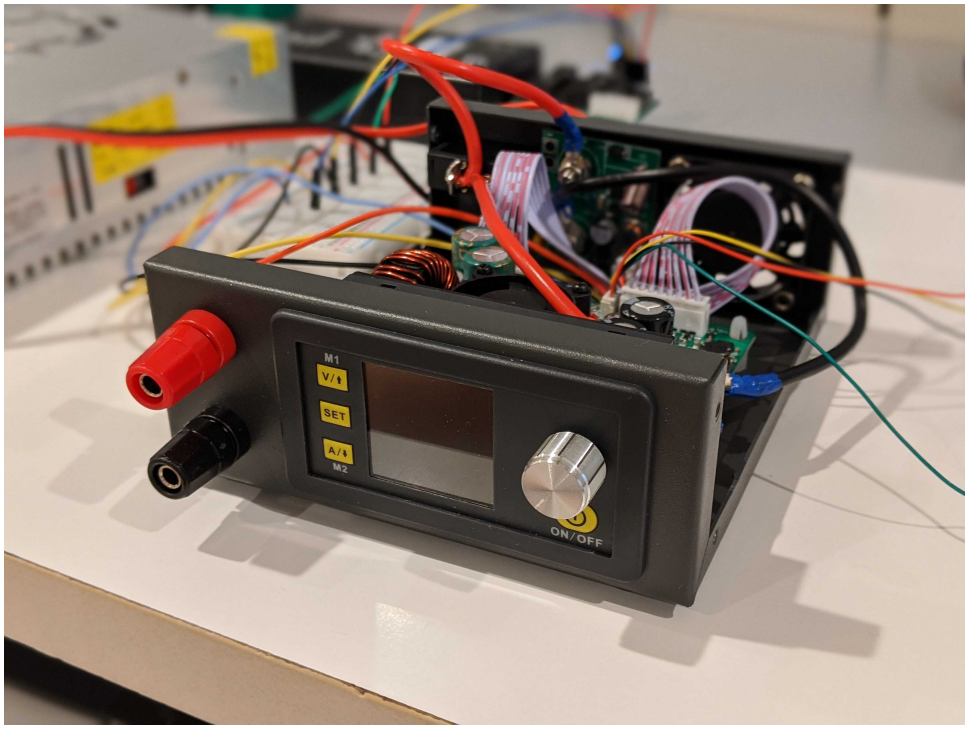

Figure 5.4.3: RD DPS5015 power supply, opened [122]

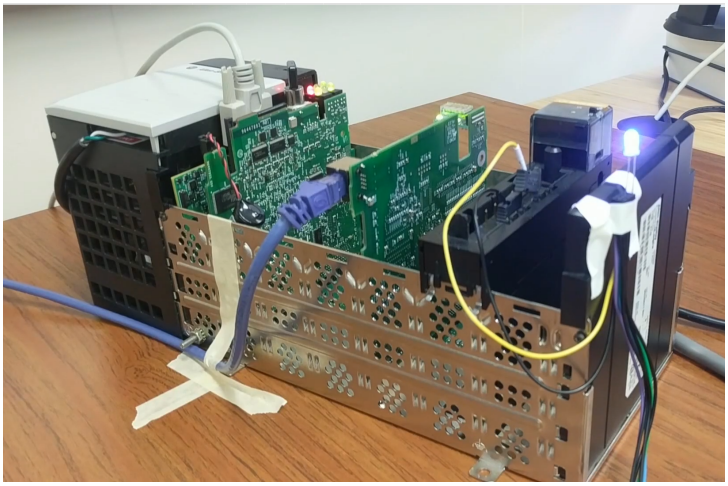

Figure 5.4.4: AB ControlLogix PLC [125], disassembled. The LED's blink is clamped in the firmware by our retrofit payload.

Chapter 6

Conclusion

Throughout this work, we have explored the challenges of finding and fixing security vulnerabilities in monolithic embedded firmware. The lack of metadata found in conventional binary programs, the intermixing of code and data, and the lack of abstractions around hardware interactions all combine to produce a unique challenge for human analysts and automated analyses alike. This creates two major gaps in the normal lifecycle of vulnerability discovery: environment modeling to enable dynamic analyses, and patching without the manufacturer's help.

The Future of Re-Hosting. We explored two different techniques to environment modeling for dynamic analysis, which is commonly termed *re-hosting*: an automated approach based on recording and automated modeling (Chapter 3), and an approach based on High-Level Emulation using analyst-provided models. These approaches have their own strengths and weaknesses; Pretender requires less analyst effort, but requires invasive device access, and works on a smaller range of devices. HALucinator, by contrast, does not require access to the device at all, and can handle numerous cases such as DMA, which Pretender cannot. However, it currently requires the analyst to build the handlers and models needed for the firmware to run. Both approaches have massively expanded the set of hardware that can be re-hosted, and created openly-accessible frameworks for

the further development of this technology.

However, the problem is far from solved. External peripherals remain one of the most complex parts of re-hosting firmware. Pretender handles external peripherals, such as the I2C temperature sensor, and RF hardware examples, but does so by modeling the on-chip peripheral and its associated external device as a composite. This makes our models specific to a given physical hardware configuration. Ideally, this would not be the case; for example, a common serial port can be thought of a simple bi-directional channel over which the CPU and the external device communicate, and we could develop models for each external serial-based peripheral using this channel alone, and reuse these on different host CPUs. However, these ports and bus controllers have their own internal hardware, which follows its own state machine, that responds to the data transferred to and from the peripheral. A particular complication is that, from the point-of-view of MMIO, it is impossible to reliably distinguish values read from control or configuration registers from data coming from outside the CPU.

With HALucinator, however, we get this separation by way of human modeling, but we still do not have a solution for modeling peripherals. Similarly, Shimware's IOFinder can help us to determine the sources of actual data in peripherals. A hybrid of these techniques may help us take this next important step in increasing the set of re-hostable devices.

The Future of Security Retrofitting. As we discuss in Chapter 5, many factors beyond the control of the analyst, can influence the patchability of a system. Interestingly, many of these mechanisms, such as hardware roots-of-trust, software signature verification, and the disabling of hardware code flashing mechanisms, are often rationalized as ways of increasing the security of the code against unauthorized modification. Indeed, the increasingly common hardware-backed integrity verification methods do cleanly accomplish this goal, while leaving some possibility of patching. For example, many boot-time

verification schemes in larger systems (e.g., PC SecureBoot and mobile phone bootloaders [128]) allow for the possibility for the user to "unlock" this chain of trust, and run their own code, at their own risk, if desired.

It is very important, however, to distinguish these techniques from those that instead aim to implement intellectual property protection, such as flash read-back protection [111, 129], which are commonly found in some form in most embedded microcontrollers. While these are also commonly touted as security measures, they only achieve this goal when paired with one of the above verification approaches. Unfortunately, for those vendors who indeed wish to add intellectual property protection measures, these methods are directly at odds with security retrofitting, as the firmware itself cannot be easily obtained.

Unless something is done to strike a balance between security, IP protection, and the end-user's ability to repair their own devices, the number of unpatched, abandoned devices will continue to increase. One solution that still allows for both kinds of protection is to implement a means of relinquishing control of these protections when the device's period of support expires. In the case of IP protection, this aligns with the fact that the vendor has no more commercial interest in the product, and therefore has no need to obfuscate their firmware further. These ideas seem a natural fit for recent and future IoT security-related legislation [130, 131] being proposed worldwide, intended to help inform users about the support and security policies of their devices.

While we hope that researchers and device vendors will work together to find and fix problems in devices, this work has taken a significant step and giving analysts and users autonomy in dealing with the security issues of their embedded devices. Monolithic device firmware can now be tested and analyzed more than before, without the source code, or the original toolset used to create it. Bugs, when found, can now be resolved more easily, even if the manufacturer can't or won't fix them quickly. Most importantly,

every possible component of the work developed here has been made available publicly, to enable its reproducibility, and to serve as a stable platform for future research. We hope these important steps will lead to a safer connected world.

Bibliography

[1] Gartner, *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*, 2015.

[2] B. Krebs, "Source Code for IoT Botnet 'Mirai' Released." https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/, October, 2016.

[3] N. Woulf, "DDoS attack that disrupted internet was largest of its kind in history, experts say." `https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet`, 2016.

[4] J. Leyden, "Mirai IoT botnet blamed for 'smashing Liberia off the internet'." `http://www.theregister.co.uk/2016/11/04/liberia_ddos/`, 2016.

[5] CVE Editorial Board, "CVE Program Status Update."

[6] C. Stupp, "Commission Plans Cybersecurity Rules for Connected Machines." https://www.euractiv.com/section/innovation-industry/news/commission-plans-cybersecurity-rules-for-internet-connected-machines/, October, 2016.

[7] S. Exchange, "What security risks does the Test Access Port (TAP) introduce?." `https://electronics.stackexchange.com/questions/253958/what-security-risks-does-the-test-access-port-tap-introduce`, 2016.

[8] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A.-R. Sadeghi, M. Maniatakos, and R. Karri, *The cybersecurity landscape in industrial control systems*, *Proceedings of the IEEE* **104** (2016), no. 5 1039–1057.

[9] 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 *Network and Distributed System Security Symposium, San Diego, CA*, 2018.

[10] 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.

[11] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, *AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares*, in *NDSS*, 2014.

[12] M. Muench, A. Francillon, and D. Balzarotti, *Avatar2: A multi-target orchestration platform*, in *BAR 2018, Workshop on Binary Analysis Research*, 2018.

[13] K. Koscher, T. Kohno, and D. Molnar, *SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems*, in *WOOT*, 2015.

[14] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, *Repeatable reverse engineering with PANDA*, in *Program Protection and Reverse Engineering Workshop*, p. 4, ACM, 2015.

[15] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster, *TARDIS: software-only system-level record and replay in wireless sensor networks*, in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 286–297, ACM, 2015.

[16] J. Nazario, "The problem with patching in addressing IoT vulnerabilities." `https://www.fastly.com/blog/problem-patching-addressing-iot-vulnerabilities`, 2017.

[17] J. Rich, "What happens when the sun sets on a smart product?." `https://www.ftc.gov/news-events/blogs/business-blog/2016/07/what-happens-when-sun-sets-smart-product`, 2016.

[18] "D-Link Adds More Buggy Router Models to 'Won't Fix' List." `https://threatpost.com/d-link-wont-fix-router-bugs/150438/`, 2019.

[19] L. H. Newman, "Decades-Old Code Is Putting Millions of Critical Devices at Risk." `https://www.wired.com/story/urgent-11-ipnet-vulnerable-devices/`, 2019.

[20] C. Cimparu, "DHS and FDA warn about much broader impact of Urgent/11 vulnerabilities." `https://www.zdnet.com/article/dhs-and-fda-warn-about-much-broader-impact-of-urgent11-vulnerabilities/`, 2019.

[21] S. Wang, P. Wang, and D. Wu, *Uroboros: Instrumenting stripped binaries with static reassembling*, in *Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[22] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, *Ramblr: Making reassembly great again.*, in *NDSS*, 2017.

[23] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, *Revarm: A platform-agnostic arm binary rewriter for security applications*, in *Annual Computer Security Applications Conference (ACSAC)*, pp. 412–424, 2017.

[24] G. Hunt and D. Brubacher, *Detours: Binary Interception of Win32 Functions*, in *3rd Usenix Windows NT Symp.*, 1999.

[25] M. Zalewski., *American fuzzy lop*, 2017. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[26] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, *SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis*, in *IEEE Symposium on Security and Privacy*, 2016.

[27] V. Chipounov, V. Kuznetsov, and G. Candea, *S2e: A platform for in-vivo multi-path analysis of software systems*, *Acm Sigplan Notices* **46** (2011), no. 3 265–278.

[28] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, *FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution*, in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pp. 463–478, 2013.

[29] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, *AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.*, in *NDSS*, 2014.

[30] 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.

[31] 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.

[32] 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.

[33] D. D. Chen, M. Egele, M. Woo, and D. Brumley, *Towards Automated Dynamic Analysis for Linux-based Embedded Firmware*, in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[34] 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)*, BAR 18, February, 2018.

[35] B. Feng, A. Mera, and L. Lu, *P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling*, in *USENIX Security Symposium*, 2020.

[36] A. Costin, A. Zarras, and A. Francillon, *Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces*, in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 437–448, ACM, 2016.

[37] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, *Cross-level sensor network simulation with cooja*, in *IEEE conference on local computer networks*, IEEE, 2006.

[38] "mbed OS." `https://www.mbed.com/en/development/mbed-os/`, 2020.

[39] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, *RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT*, *IEEE Internet of Things Journal* (2018).

[40] "UltraHLE." `https://en.wikipedia.org/wiki/UltraHLE`, 2019.

[41] "Dolphin Emulator." `https://dolphin-emu.org/`, 2019.

[42] N. Corteggiani, G. Camurati, and A. Francillon, *Inception: System-wide security testing of real-world embedded systems software*, in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), USENIX Association, August, 2018.

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

[44] J. Zaddach, D. Balzarotti, and A. Francillon, *Development of novel dynamic binary analysis techniques for the security analysis of embedded devices*. PhD thesis, TELECOM ParisTech, 2015.

[45] 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)*, USENIX Association, 2018.

[46] 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.

[47] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, *Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.*, in *NDSS*, 2015.

[48] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler, *Firmusb: Vetting usb device firmware using domain informed symbolic execution*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2245–2262, ACM, 2017.

[49] Osbourne, Paul, "Cmsis-svd repository and parsers." `https://github.com/posborne/cmsis-svd`.

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

[51] A. Beckus, *Qemu with an stm32 microcontroller implementation*, 2012. `http://beckus.github.io/qemu_stm32/`.

[52] L. Ionescu, *Gnu mcu eclipse. a family of eclipse cdt extensions and tools for gnu arm & risc-v development*, 2015. `https://gnu-mcu-eclipse.github.io/`.

[53] Comsecuris, "Luaqemu." `https://github.com/comsecuris/luaqemu`.

[54] D. Thomas and R. Rolf, *Graph-based comparison of executable objects*, in *Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications, ser. SSTIC*, vol. 5, 2005.

[55] H. Flake, *Structural comparison of executable objects*, in *Proc. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pp. 161–174, 2004.

[56] "Diaphora: A Free and Open Source Program Diffing Tool." `http://diaphora.re/`.

[57] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, *Byteweight: Learning to recognize functions in binary code*, in *USENIX Security Symp.*, 2014.

[58] I. Guilfanov, " Fast Library Identification and Recognition Technology." `https://www.hex-rays.com/products/ida/tech/flirt/index.shtml`.

[59] E. R. Jacobson, N. Rosenblum, and B. P. Miller, *Labeling library functions in stripped binaries*, in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ACM, 2011.

[60] J. Qiu, X. Su, and P. Ma, *Using reduced execution flow graph to identify library functions in binary code*, *IEEE Transactions on Software Engineering* **42** (2016), no. 2 187–202.

[61] J. Qiu, X. Su, and P. Ma, *Library functions identification in binary code by using graph isomorphism testings*, in *IEEE Conf. on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2015.

[62] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, *Fossil: A resilient and efficient system for identifying foss functions in malware binaries*, *ACM Transactions on Privacy and Security (TOPS)* **21** (2018), no. 2 8.

[63] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, *Debin: Predicting debug information in stripped binaries*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 1667–1680, ACM, 2018.

[64] T. Dullien, "Searching statically-linked vulnerable library functions in executable code." `https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html`.

[65] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, *From Hack to Elaborate Technique - A Survey on Binary Rewriting*, in *ACM Computing Surveys (CSUR)*, 2019.

[66] S. Wang, P. Wang, and D. Wu, *Reassembleable disassembling.*, in *USENIX Security*, pp. 627–642, 2015.

[67] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, *Bintrimmer: Towards static binary debloating through abstract interpretation*, in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 482–501, 2019.

[68] E. Bauman, Z. Lin, and K. W. Hamlen, *Superset disassembly: Statically rewriting x86 binaries without heuristics.*, in *NDSS*, 2018.

[69] S. Dinesh, N. Burow, D. Xu, and M. Payer, *RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization*, in *IEEE Security and Privacy*, 2020.

[70] S. Wang, W. Wang, Q. Bao, P. Wang, X. Wang, and D. Wu, *Binary code retrofitting and hardening using sgx*, in *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[71] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, *Automating patching of vulnerable open-source software versions in application binaries.*, in *NDSS*, 2019.

[72] Y. Hu, Y. Zhang, and D. Gu, *Automatically patching vulnerabilities of binary programs via code transfer from correct versions*, *IEEE Access* **7** (2019) 28170–28184.

[73] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, *Cimplifier: automatically debloating containers*, in *Joint Meeting on Foundations of Software Engineering*, pp. 476–486, 2017.

[74] A. Quach and A. Prakash, *Bloat factors and binary specialization*, in *ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019.

[75] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, *Nibbler: debloating binary shared libraries*, in *Annual Computer Security Applications Conference*, pp. 70–83, 2019.

[76] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, *Razor: A framework for post-deployment software debloating*, in *USENIX Security Symposium*, 2019.

[77] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et. al.*, *A density-based algorithm for discovering clusters in large spatial databases with noise.*, in *Conference on Knowledge Discovery and Data Mining*, 1996.

[78] Maxim Integrated, *MAX32600MBED ARM mbed Enabled Development Platform for MAX32600*, 2018. `https://www.maximintegrated.com/en/products/microcontrollers/MAX32600MBED.html`.

[79] STMicroelectronics, *STM32F072RB*, 2018. `https://www.st.com/en/microcontrollers/stm32f072rb.html`.

[80] R. Toulson and T. Wilmshurst, *Fast and effective embedded systems design: applying the ARM mbed*. Newnes, 2016.

[81] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, *Difuze: Interface aware fuzzing for kernel drivers*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 2123–2138, ACM, 2017.

[82] "CSAW Embedded Security Challenge." `https://csaw.engineering.nyu.edu/esc`, 2019.

[83] "TrustworthyComputing / csaw_esc_2019 - Github." `https://github.com/TrustworthyComputing/csaw_esc_2019`, 2019.

[84] A. Dunkels, B. Gronvall, and T. Voigt, *Contiki: a lightweight and flexible operating system for tiny networked sensors*, in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pp. 455–462, IEEE, 2004.

[85] "STM32Cube MCU Packages."
https://www.st.com/en/embedded-software/stm32cube-mcu-packages.html.

[86] "MCUXpresso Software Development Kit (SDK)."
https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-software-development-kit-sdk:MCUXpresso-SDK.

[87] "Atmel Advanced Software Framework."
http://asf.atmel.com/docs/latest/architecture.html.

[88] "System Workbench for STM32." https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-ides/sw4stm32.html.

[89] "Code composer studio integrated development environment."
http://www.ti.com/tool/CCSTUDIO.

[90] "MCUXpresso Integrated Development Environment (IDE)."
https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE.

[91] "Atmel studio 7 - microchip technologies."
https://www.microchip.com/mplab/avr-support/atmel-studio-7.

[92] "Amazon FreeRtOS Vendors."
https://github.com/aws/amazon-freertos/tree/master/vendors.

[93] "Mbed OS Repo - ARMmbed/mbed-os/targets."
https://github.com/ARMmbed/mbed-os/tree/master/targets.

[94] "stm32duino - Arduino_Core_STM32/system."
https://github.com/stm32duino/Arduino_Core_STM32/tree/master/system.

[95] "Build With Mbed." https://www.mbed.com/built-with-mbed/.

[96] "lwIP - A Lightweight TCP/IP stack."
http://savannah.nongnu.org/projects/lwip.

[97] "The FreeRTOS Kernel." https://www.freertos.org/.

[98] F. Bellard, *QEMU, a fast and portable dynamic translator*, in *USENIX Annual Technical Conference*, vol. 41, p. 46, 2005.

[99] T. Instruments, "Code Composer Studio (CCS) Integrated Development Environment (IDE." http://www.ti.com/tool/CCSTUDIO, 2019.

[100] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, *AddressSanitizer: A Fast Address Sanity Checker*, in *USENIX Annual Technical Conference*, pp. 309–318, 2012.

[101] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, *SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis*, in *IEEE Symposium on Security and Privacy*, 2016.

[102] "HALucinator: stm32f4_uart.py." https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/stm32f4/stm32f4_uart.py, 2019.

[103] "Halucinator: rf233.py." https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/atmel_asf_v3/rf233.py, 2019.

[104] "ZeroMQ: Distributed Messaging." http://zeromq.org/.

[105] "AFL-Unicorn." https://github.com/Battelle/afl-unicorn.

[106] "STM32479I-EVAL." http://www.st.com/resource/en/user_manual/dm00219329.pdf.

[107] "STM NUCLEO-F401RE Development Board." https://www.st.com/en/evaluation-tools/nucleo-f401re.html.

[108] "SAM R21 Xplained Pro User Guide." http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42243-SAMR21-Xplained-Pro_User-Guide.pdf.

[109] "FRDM-K64F Platform." https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F.

[110] "FatFs: Generic FAT Filesystem Module." http://elm-chan.org/fsw/ff/00index_e.html.

[111] STMicroelectronics, "AN4701: Proprietary code read-out protection on microcontrollers of the STM32F4 Series." https://www.st.com/resource/en/application_note/dm00186528-proprietary-code-readout-protection-on-microcontrollers-of-the-stm32f4-series-stmicroelectronics.pdf, 2020.

[112] F. Armstrong, "A Discussion on Atmel Lock Byte and Firmware Protection." https://www.avrfreaks.net/sites/default/files/A%20discussion%20on%20Atmel%20Lock%20Bits.pdf, 2013.

[113] TI, "Understanding security features for MSP430™ Microcontrollers." http://www.ti.com/lit/ml/swpb018/swpb018.pdf?ts=1587844615741, 2020.

[114] J. Obermaier and S. Tatschner, *Shedding too much light on a microcontroller's firmware protection*, in *USENIX WOOT*, 2017.

[115] M. Schink and J. Obermaier, "Exception(al) Failure - Breaking the STM32F1 Read-Out Protection." `https://blog.zapb.de/stm32f1-exceptional-failure/`, 2020.

[116] S. Vasile, D. Oswald, and T. Chothia, *Breaking all the things — a systematic survey of firmware extraction techniques for iot devices*, in *International Conference on Smart Card Research and Advanced Applications*, pp. 171–185, 2018.

[117] F. Gröbert, *Automatic identification of cryptographic primitives in software*, *Ruhr-University Bochum* (2010) 115.

[118] P. Lestringant, F. Guihéry, and P.-A. Fouque, *Automated identification of cryptographic primitives in binary code with data flow graph isomorphism*, in *ACM Symposium on Information, Computer and Communications Security (CCS)*, 2015.

[119] subwire, "Autoblob: Automatic Blob-loading for CLE." `https://github.com/subwire/autoblob`, 2020.

[120] J. Lee, T. Avgerinos, and D. Brumley, *Tie: Principled reverse engineering of types in binary programs.*, in *NDSS*, 2011.

[121] 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 *IEEE Security & Privacy*, 2020.

[122] AliExpress, "RD DPS5015." `https://www.aliexpress.com/item/32702714880.html`, 2020.

[123] EEVBlog, "Flaming Power Supply!." `https://www.youtube.com/watch?v=Q2rvAoO-MIA`, 2017.

[124] "OpenDPS." `https://github.com/kanflo/opendps`, 2020.

[125] "ControlLogix and GuardLogix Controllers, author=Rockwell Automation, howpublished=`https://literature.rockwellautomation.com/idc/groups/literature/documents/td/1756-td001_-en-p.pdf`, year=2020."

[126] "Arduino." `http://arduino.cc/`, 2020.

[127] S. H. Ding, B. C. Fung, and P. Charland, *Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization*, in *IEEE Security and Privacy*, 2019.

[128] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Bootstomp: On the security of bootloaders in mobile devices*, in *USENIX Security Symposium*, (Vancouver, BC), 2017.

[129] Microchip, "AT16743: SAM V7/E7/S7 Safe and Secure Bootloader." http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42725-Safe-and-Secure-Bootloader-for-SAM-V7-E7-S7-MCUs_AT16743_ApplicationNote.pdf.

[130] "CA S.B. 327." https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180SB327, 2020.

[131] "S.734 - Internet of Things Cybersecurity Improvement Act of 2019." https://www.congress.gov/bill/116th-congress/senate-bill/734, 2020.