Toward the Analysis of Embedded Firmware through Automated Re-hosting

Eric Gustafson[1,2]*, Marius Muench*[3]*, Chad Spensky*[1]*, Nilo Redini*[1]*, Aravind Machiry*[1]*, Yanick Fratantonio*[3]
Aurélien Francillon[3]*, Davide Balzarotti*[3]*, Yung Ryn Choe*[2]*, Christopher Kruegel*[1]*, and Giovanni Vigna*[1]
[1]*University of California, Santa Barbara*
{edg, cspensky, nredini, machiry, chris, vigna}@cs.ucsb.edu
[2]*Sandia National Laboratories*
{edgusta, yrchoe}@sandia.gov
[3]*EURECOM*
{marius.muench, francill, yanick.fratantonio, balzarot}@eurecom.fr

Abstract

The recent paradigm shift introduced by the Internet of Things (IoT) has brought embedded systems into focus as a target for both security analysts and malicious adversaries. Typified by their lack of standardized hardware, diverse software, and opaque functionality, IoT devices present unique challenges to security analysts due to the tight coupling between their firmware and the hardware for which it was designed. In order to take advantage of modern program analysis techniques, such as fuzzing or symbolic execution, with any kind of scale or depth, analysts must have the ability to execute firmware code in emulated (or virtualized) environments. However, these emulation environments are rarely available and are cumbersome to create through manual reverse engineering, greatly limiting the analysis of binary firmware.

In this work, we explore the problem of *firmware re-hosting*, the process by which firmware is migrated from its original hardware environment into a virtualized one. We show that an approach capable of creating virtual, interactive environments in an automated manner is a necessity to enable firmware analysis at scale. We present the first proof-of-concept system aiming to achieve this goal, called PRETENDER, which uses observations of the interactions between the original hardware and the firmware to automatically create models of peripherals, and allows for the execution of the firmware in a fully-emulated environment. Unlike previous approaches, these models are interactive, stateful, and transferable, meaning they are designed to allow the program to receive and process new input, a requirement of many analyses. We demonstrate our approach on multiple hardware platforms and firmware samples, and show that the models are flexible enough to allow for virtualized code execution, the exploration of new code paths, and the identification of security vulnerabilities.

1 Introduction

The new wave of commercialized embedded systems, brought about by trends such as the IoT, has resulted in their use for an increasing number of security and safety-critical applications. The most unusual feature of this new computing paradigm is its extreme diversity, in terms of both hardware and software. 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.

Unfortunately for security researchers, 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.

Emulators for these systems are a key component in enabling dynamic analysis of the firmware at scale, as transparent on-device analysis is rarely possible, and it is impractical to acquire hundreds of identical physical devices to parallelize the analysis process. However, appropriate emulators are typically unavailable, particularly due to the impracticality of properly supporting the thousands of incompatible embedded CPUs, and an enormous selection of external peripherals. Worse yet, the physicality of these devices means that analyzing their firmware without the sensors, actuators, and other components may not be useful, or even possible at all.

Previous efforts have avoided the problem through the use of an operating system abstraction [3, 8], or with a hardware-in-the-loop scheme [15, 16, 26]. However, these techniques impose severe limits on the scale and scope of analyzable targets, such as requiring that a general-purpose OS is present, or a significant amount of potentially costly original hardware to be tractable. Without these approaches, analysts must manually implement models of all the on-chip and off-chip peripherals for a device. This requires that the analyst can obtain complete documentation or thorough understanding for every component of the system, and spends the time

to manually develop components usable by the emulator. Manufacturers can also use completely custom components, for which no documentation can be obtained, rendering emulation by any existing method extremely difficult.

We explore the possibility of automated *firmware re-hosting*. The key idea behind firmware re-hosting consists of analyzing a given firmware/hardware combination (possibly through multiple execution rounds), understanding what the firmware expects from the surrounding hardware, and then attempting to *replace the hardware altogether, so that the firmware analysis can be carried out with software-only components*. In essence, firmware re-hosting would allow analysts to decouple the execution of firmware from the hardware on which it expects to be executed. This allows for the scaling of popular dynamic analysis techniques, outperforming hardware-in-the-loop or device-only approaches [20].

We identified four key aspects that are necessary for building a re-hosting solution to deal with today's embedded firmware: 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.

In this work, we develop an approach to re-hosting that achieves all of them, 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 [2]) or program analysis engines (e.g., `angr` [23]) 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 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 techniques, 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.

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 [27] or symbolic execution engines such as angr [23] or S2E [4], rely on some

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

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 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 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.
- **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.

Table 1: Excerpt of tools tackling the re-hosting problem

Tool	Virtual	Interactive	Abstraction-less	Automatic
Simics [17]	✓	✓	✓	-
FIE [9]	✓	✓	✓	-
Avatar [26]	-	✓	✓	-
PROSPECT [14, 15]	-	✓	-	✓
Surrogates [16]	-	✓	✓	-
Firmadyne [3]	✓	✓	-	✓
Avatar^2 [19]	✓	✓	✓	-
PRETENDER	✓	✓	✓	✓

2.1 Re-hosting Aspects and Related Work

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 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.

Virtual. A re-hosting solution should not depend on the presence of hardware during analysis. Many proposed approaches to firmware analysis [7, 15, 16, 26] 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 [20]. 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.

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 [10], 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.

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 [3, 8]. 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.

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 [17]), 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 [9, 12, 22] 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 [21], 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 Section 4), 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 [2] 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* [1] or *GNU MCU Eclipse* [13]. A different approach is taken by *LuaQEMU* [5] and *avatar*^{2} [19], which provide an interface for the analyst to define the peripheral

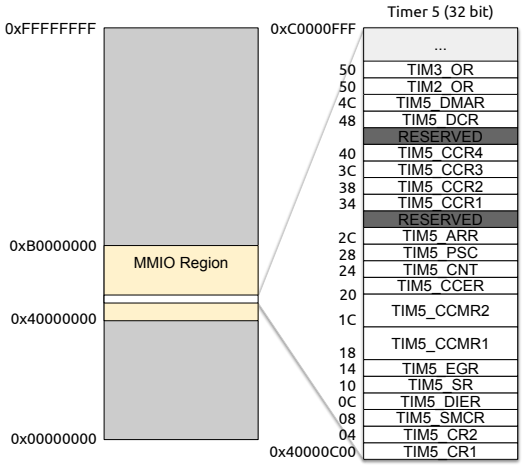

Figure 1: The memory layout for a simple 32 bit memory-mapped timer on the STM32 embedded processor.

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.

While we, of course, do not claim to have achieved the goal of ideal re-hosting in this work, in the following sections, we will showcase a proof-of-concept approach that has all of the above properties, with limitations discussed in Section 5.

3 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 5.

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 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

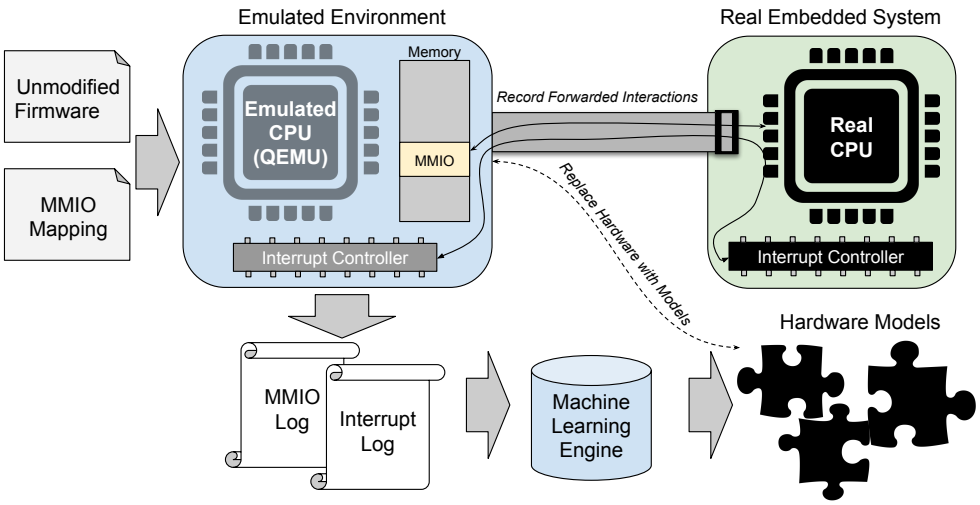

Figure 2: Overview of the functionality of PRETENDER

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 5. 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 2. In the remainder of this section, we will discuss the individual phases of the system in detail.

3.1 Recording

On ARM-based platforms, MMIO accesses occur through normal load or store instructions from the CPU, and take place across the CPU's internal memory buses. Since we cannot observe this activity directly, or either via a debugger or through physical access, we can instead effectively extend the memory bus outside the chip where the data required for modeling can be recorded. To this end, we leverage 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 [19], 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. All extensions and modifications to *avatar*^{2} developed during this

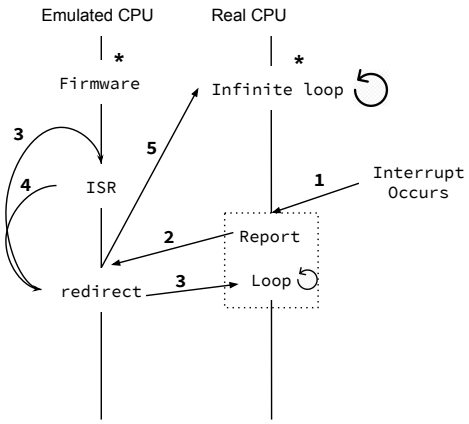

Figure 3: State diagram of interrupt recording in PRETENDER. * indicates the initial state.

work will be released as open-source alongside with the code of PRETENDER upon the publication of this paper.

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.*[2] Figure 3 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).

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.

3.2 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 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, as well as the manuals for all of the devices explored in Section 4. 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 [11] 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 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.

3.3 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

[2] Recording interrupts is a particularly complex matter, requiring precise synchronization of the emulator and hardware to avoid incorrect behavior. We detail the problem and the rationale behind our approach in Appendix A.

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 4.

3.4 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.

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.[3]

Test Harness Creation. Finally, in order for this system to be fully *interactive*, as we discuss in Section 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 *avatar*^{2}'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 4, we created a harness consisting of feeding input data via the device's serial port.

[3] For a walk-through of the state approximation model in action and the challenges faced by it, see Appendix B.

4 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 [18] and the STM Nucleo F072RB [24]. 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` [25] 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 analyst 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

Table 2: 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

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 [3, 8], 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 2 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 source of input. A

Table 3: 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

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 [6] 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 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 shows a memory trace acquired by PRETENDER, and shows interactions with the timer (Table 3a) and the flash memory controller (Table 3b). 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 functionality 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. 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.[4] Instead, we reuse

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

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 2.

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 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.

5 Discussion and Future Work

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, 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, however. 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 [7].

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.

External Peripherals. 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. Separating these two intertwined systems remains an important, open problem.

Heavily-stateful Peripherals. Not all peripherals, particularly external ones, are well-modeled by a state machine. As we discussed in Section 3, 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.

Adding Abstractions. While a system that is abstraction-less is the most ideal solution to the re-hosting problem, modeling using a higher-level abstraction, such as libraries or an OS, remains an important way to make re-hosting more robust. Many firmware images, including the ones used in this work, were written with such libraries, which perform most hardware interactions on behalf of the author's code. If located, these would also provide a convenient means of dealing with the above problem of external peripherals and DMA, as they provide the firmware author a high-level way of communicating with peripherals, which can then be exploited for modeling. However, for firmware without an operating system, which is typically distributed as a binary blob, this reduces to the problem of identifying library functions in statically-compiled, stripped binary programs, a well-studied but yet-unsolved problem. Furthermore, any code which violates the abstraction by controlling hardware directly still requires the use of a technique like PRETENDER. This is found even in our simple examples, where all accesses to the GPIO peripheral were aggressively in-lined by the compiler, such that no library call or other abstraction remained.

6 Conclusion

In this work, 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.

Acknowledgements

Sandia National Laboratories is 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.

This material 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. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

Additionally, this work was in part funded by a research contract with Siemens AG.

References

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

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

[3] 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.

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

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

[6] 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*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2123–2138. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134069

[7] 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, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[8] 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*. ACM, 2016, pp. 437–448.

[9] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution." in *USENIX Security*, 2013, pp. 463–478.

[10] 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*. ACM, 2015, p. 4.

[11] 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.

[12] 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*. ACM, 2017, pp. 2245–2262.

[13] 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/.

[14] 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.

[15] 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*. ACM, 2014, pp. 329–340.

[16] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems." in *WOOT*, 2015.

[17] 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*, vol. 35, no. 2, pp. 50–58, 2002.

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

[19] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A Multi-target Orchestration Platform," in *Workshop on Binary Analysis Research (colocated with NDSS Symposium)*, ser. BAR 18, February 2018.

[20] 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*, ser. NDSS 18, February 2018.

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

[22] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.

[23] 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.

[24] STMicroelectronics, "STM32F072RB," 2018, https://www.st.com/en/microcontrollers/stm32f072rb.html.

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

[26] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, 2014.

[27] M. Zalewski., "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.

Appendices

A Recording Rationale

While we describe our means of recording in Section 3.1, our approach may seem overly complicated. In the following, we point out the rationale behind the design decisions for the recording subcomponent of PRETENDER.

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, 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.

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). As a result of these complications, our approach involves virtually extending the internal memory bus of the device, by emulating the firmware, and forwarding and recording only the hardware-related accesses to the original physical device (as detailed in Section 3).

Recording Interrupts. 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 [26] or SURROGATES [16] 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. [7], 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 *avatar*^{2} 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 *avatar*^{2} 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 *avatar*^{2} 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.

B State Approximation Details

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 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. In avoiding these limitations, we created the State Approximation algorithm we describe in Section 3.

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 4. 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 B.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, 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

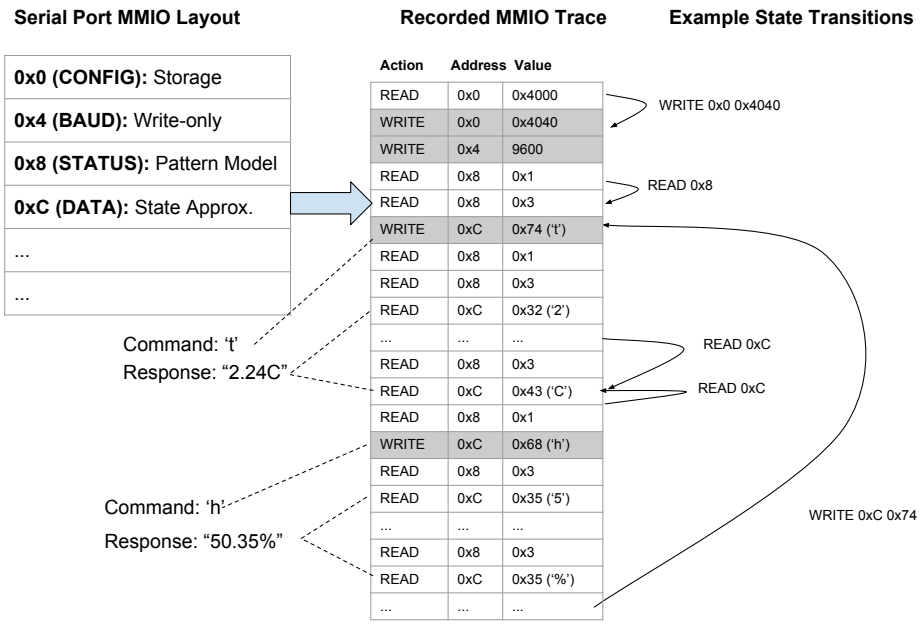

Figure B.1: Illustration of State Approximation in action, on a simplified serial port peripheral

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.