SAND2019-14482C

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation

Abraham A. Clements[*,1], Eric Gustafson[*,1,2], Tobias Scharnowski[3], Paul Grosen[2], David Fritz[1], Christopher Kruegel[2], Giovanni Vigna[2], Saurabh Bagchi[4], and Mathias Payer[5]
[1]Sandia National Laboratories, [2]UC Santa Barbara, [3]Ruhr-Universitt Bochum,
[4]Purdue University, [5]École Polytechnique Fédérale de Lausanne
{*aacleme, djfritz*}*@sandia.gov*, {*edg, pcgrosen, chris, vigna*}*@cs.ucsb.edu*, *sbagchi@purdue.edu*, *mathias.payer@epfl.ch*

Abstract

Given the increasing ubiquity of online embedded devices, analyzing their firmware is important to security, privacy, and safety. The tight coupling between hardware and firmware and the diversity found in embedded systems makes it hard to perform dynamic analysis on firmware. However, firmware developers regularly develop code using abstractions, such as Hardware Abstraction Layers (HALs), to simplify their job. We leverage such abstractions as the basis for the re-hosting and analysis of firmware. By providing high-level replacements for HAL functions (a process termed *High-Level Emulation – HLE*), we decouple the hardware from the firmware. This approach works by first locating the library functions in a firmware sample, through binary analysis, and then providing generic implementations of these functions in a full-system emulator.

We present these ideas in a prototype system, HALucinator, able to re-host firmware, and allow the virtual device to be used normally. First, we introduce extensions to existing library matching techniques that are needed to identify library functions in binary firmware, to reduce collisions, and for inferring additional function names. Next, we demonstrate the re-hosting process, through the use of simplified *handlers* and *peripheral models*, which make the process fast, flexible, and portable between firmware samples and chip vendors. Finally, we demonstrate the practicality of HLE for security analysis, by supplementing HALucinator with AFL, to locate multiple previously-unknown vulnerabilities in firmware middleware libraries.

1 Introduction

Embedded systems are pervasive in modern life: vehicles, communication systems, home automation systems, and even pet toys are all controlled through embedded CPUs. Increasingly, these devices are connected to the Internet for extra functionality. This connectivity introduces new security, privacy, and reliability concerns. Unfortunately, auditing the firmware of these systems is a cumbersome, time-consuming, per-device effort.

Today, 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) [26, 40], 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.

Emulation, also known as firmware re-hosting, provides a means of addressing many of these challenges, by offering the ability to execute firmware at scale through the use of commodity computers, and providing more insight into the execution than is possible on a physical device [44]. Yet, heterogeneity in embedded hardware poses a significant barrier to the useful emulation 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. In-

[*] These authors contributed equally to this work.

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.

tel's SIMICS [57, 38] 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 emulation of diverse hardware rely on a real specimen of the device, where the emulator forwards interactions with unsupported peripherals to the hardware [58, 43, 36]. 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 [22, 54, 32] focus on recording and subsequently replaying or modeling 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.

The immense diversity of hardware also affects firmware developers. 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 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 [39] 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.

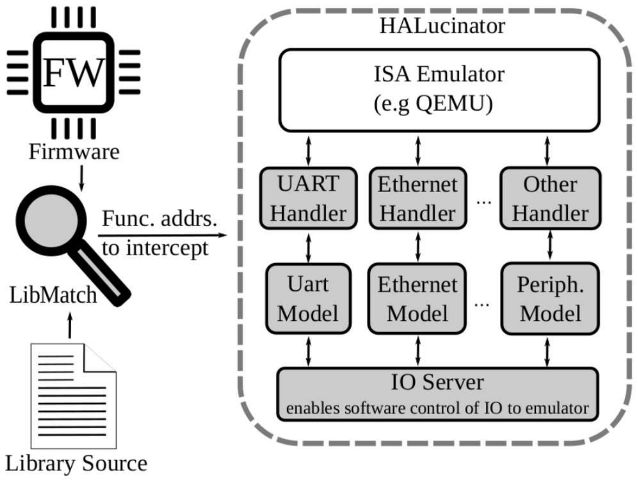

Figure 1: Overview of HALucinator, with our contribution shown in gray.

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 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 [5, 11]. In summary, our contributions 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 [25].

2 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 [53, 42, 16]. HALs provide a common abstraction for families of microcontrollers, thus a single HAL covers many different microcontrollers. For example,

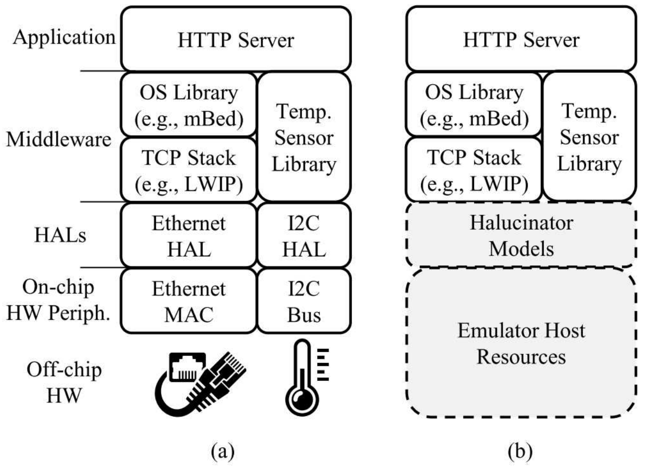

Figure 2: (a) Software and hardware stack for an illustrative HTTP Server. (b) Conceptual illustration of HTTP Server when executing using HALucinator.

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 [10, 4, 41, 2] and third party development tools (*e.g.,* Keil, IAR). These same HALs are included in embedded OSes (*e.g.,* in FreeRTOS [1], mBed OS [8], RIOT OS [17], and Arduino [9]). These OSes are currently used in commercially available devices [3]. 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 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 5 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.

2.1 Emulating Hardware and Peripherals

To achieve our goal of scalably re-hosting embedded firmware, we must emulate the environment it runs in. This environment consists of, first and foremost, the main CPU of the device with its instruction set and basic memory features. Modern CPUs, even low-power, low-cost microcontrollers, include a full complement of on-chip peripherals, including timers, bus controllers, networking components, and display devices. Code executing on the CPU controls these features via Memory-Mapped I/O (MMIO), where various control and data registers of peripherals are accessed as normal memory locations in a pre-determined region. The exact layout and semantics of each peripheral's MMIO regions vary, but are described in the chip's documentation.

Further complicating re-hosting is the interaction of a firmware with off-chip devices (*e.g.,* sensors, actuators, external storage devices, or communications hardware). As each product usually contains custom-designed circuit boards, the complete execution environment of each firmware sample is largely unique. Existing emulation tools (*e.g.,* QEMU [18] and SIMICS [57]) support a relatively limited number of CPU's, and even fewer on-chip and off-chip devices. To use these tools, the on-chip and off-chip devices must be implemented to conform to the MMIO register interface used by the firmware. This requires understanding and implementing the state machines and logic of each device, a time consuming and challenging task.

2.2 The Firmware Stack

The software and hardware stack for an illustrative HTTP server is shown in Figure 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) [37]. 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 [39], FreeRTOS [30], and Contiki [25]), 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 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 5. 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.

2.3 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 [53]. 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.

3 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 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 Section 3.2, 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.

3.1 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 emulation 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 7, 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 [18] faithfully emulate instructions. Each Cortex-M vendor provides open-source HAL(s) for their chips, with compilers and configurations [53, 16, 42, 34]. 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.

3.2 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 [33, 35, 24] 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 [28, 21], 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 6), 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 5.

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 [28, 21] 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.

3.3 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 5, 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 7.

3.4 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 [13]—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 terminate. 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 [44]. 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 [49], 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 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` [50] 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 Avatar^2 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 Avatar^2 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 [7] and [6].

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 [59] 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 5.

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 [14]. 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.

5 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 [52], STM32-Nucleo F401RE [51], SAM R21 Xplained Pro [48], NXP FRDM-K64F [29]) 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 [53], NXP's MCUXpresso [42], Atmel's Advanced Software Framework (ASF) [16], lwIP [37], FatFS [27], and Contiki-OS [25], 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.

5.1 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 1. LibMatch without context matching is comparable to what is achievable with current matching algorithms (*e.g.,* BinDiff [28], or Diaphora [21]). 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 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 context 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 can-

Mfg.	Application	HAL	LibMatch Without Context Matching				LibMatch With Context Matching				
		Syms	Correct	Collision	Incorrect	Missing	Correct	Collision	Incorrect	Missing	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 Receiver	299	224 (74.9%)	63	2	10	273 (91.3%)	17	4	5	24
Atmel	6LoWPAN Sender	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 Server	235	188 (80.0%)	43	0	4	207 (88.1%)	24	0	0	6
STM	UDP Echo Client	235	186 (79.1%)	43	0	4	205 (87.2%)	24	0	0	8
STM	TCP Echo Server	239	192 (80.3%)	43	0	4	211 (88.3%)	24	0	0	5
STM	TCP Echo Client	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 Server	170	133 (78.2%)	25	0	12	141 (83.0%)	16	8	5	22
NXP	TCP Echo Server	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 1: LibMatch performance, with and without contextual matching.

not 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.

5.2 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 Section A.1), 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 2 was created by taking the union of the handlers executed during interactive emulation for the binaries in Table 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

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

peripheral that enables 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 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.

5.3 Interactive Emulation Comparison

Next we re-host the 16 firmware samples shown in Table 1 interactively, using QEMU, Avatar^2 [43], 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 functionality 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 Section 5.4 HLE enables targeting the fuzzer at different layers within a firmware.

Table 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 Avatar^2, we report the number of reads and writes forwarded to the board ("Fwd R/W") which demonstrate that Avatar^2 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

				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 Sender	ASF, Contiki, uIPv6, 6LoWPAN	UART, 802.15.4, EXTI, Clock, Timer, EDBG	14	✗	121	521	✗	2,734	21	36	✓
Atmel	6LoWPAN Receiver	ASF, Contiki, uIPv6, 6LoWPAN	UART, 802.15.4, EXTI, Clock, Timer, EDBG	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 Client	STM32Cube, lwIP	Ethernet, Clock, GPIO, EXTI	8	✗	32	15	✗	732	16	10	✓
STM	UDP Echo Server	STM32Cube, lwIP	Ethernet, Clock	8	✗	40	17	✗	568	15	10	✓
STM	TCP Echo Client	STM32Cube, lwIP	Ethernet, Clock, GPIO, EXTI	8	✗	31	15	✗	1,110	16	10	✓
STM	TCP Echo Server	STM32Cube, lwIP	Ethernet, Clock	8	✗	33	15	✗	1,002	15	10	✓
STM	PLC	STM32Cube, lwIP, STM-WiFi	Clock, Timer, STM-WiFI, UART, SPI	39	✗	54	17	✗	713	17	41	✓
NXP	UART	MCUExpresso	UART	4	✗	107	1,766	✓	82	6	28	✓
NXP	UDP Echo Server	MCUExpresso, lwIP	UART, Ethernet	4	✗	54	66	✗	805	13	43	✓
NXP	TCP Echo Server	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 3: Comparison of QEMU, Avatar^2, and HALucinator.

our baseline approaches, the NXP UART firmware using Avatar^2 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). Avatar^2'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 Avatar^2 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 Avatar^2, 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.

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 Sender	1d:10	1,876,531	23,982	0
Atmel 6LoWPAN Receiver	1d:10	2,306,569	38,788	3
STM UDP Server	3d:8h	19,214,779	3,261	0
STM UDP Client	3d:8h	12,703,448	3,794	0
STM TCP Server	3d:8h	16,356,129	4,848	0
STM TCP Client	3d:8h	16,723,950	5,012	0
STM ST-PLC	1d:10h	456,368	772	27
NXP TCP Server	14d:0h	218,214,107	5164	0
NXP UDP Server	14d:0h	240,720,229	3032	0
NXP HTTP Server	14d:0h	186,839,871	9710	0

Table 4: Fuzzing experiments results.

5.4 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, and the firmware used in the experiments in WYCINWYC [44]. 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 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 [44], 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 Section 3.4. 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.

6 Related Work

HALucinator draws upon related work in function and library labeling, as well as firmware emulation.

Function Identification and Labeling. 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* [55, 28], and its open source counterpart *Diaphora* [21] 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.

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 [31], backward slices from system calls [35], and traces from symbolic execution [47, 46]. Matching the extracted features has been performed through Bayesian networks [15], neural networks [33], and locality-sensitive hashing [24]. 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.

Firmware Emulation. Many previous works have explored the challenge of emulating embedded firmware. The most prevalent approach employs hardware-in-the-loop execution, as found in AVATAR [58], AVATAR^2 [43], and SURROGATES [36]. In these systems, the physical target device is tethered to the analysis environment, typically using a debug port, and its hardware peripherals are used by a standard emulator during execution. This approach is limited by its visibility into the hardware; even with full debugger support, only the state of the processor is accessible to the emulator. State internal to peripherals is not synchronized with the emulator, and external events (*e.g.,* timeouts or data reception) modify the peripheral's state, causing it to deviate from the emulator's state, which may lead to incorrect execution or faults when the emulator attempts to modify the peripherals state by read/writing the peripherals registers. In addition, current hardware-in-the-loop approaches do not support interrupts or direct memory access (DMA). HALucinator handles interrupts and DMA through the same HALs developers use to perform DMA; enabling emulating firmware which current hardware-in-the loop approaches cannot.

Another approach [19, 20] to emulation involves using the presence of a high-level operating system, such as Linux, as a point of abstraction, and replacing the firmware's version with one able to be run in an emulator. This could be thought of as a form of high-level emulation, as it uses the user-kernel barrier as the modeling boundary. However, it only works on firmware with a file-system image which can be booted without any device-specific code being run. In this work, we specifically target "blob" firmware, found in devices without such an operating system.

All of these systems, including HALucinator, rely on an underlying emulator to execute code and provide real or emulated peripherals. The popular open-source QEMU [18] provides the basis for most, and itself includes support for a range of chips and the on-board peripheral models needed to boot some firmware. SIMICS [57, 38] allows one to implement cycle-accurate emulators, but requires tedious manual effort to build the models of any device not represented in its default distribution. However, as the number of popular embedded CPUs has exploded, the usefulness of these emulators in re-hosting a given firmware is decreasing.

HALucinator draws some inspiration from the work done in game console emulation [56, 23], which pioneered the idea of HLE, 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.

Firmware can also be re-hosted without full emulation, if source code is available. Simulators for Contiki [45], mBed [39] and RIOT-OS [17] 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.

Recently, approaches such as P2IM [12] and Pretender [32], both concurrent with this work, achieve automated re-hosting of embedded firmware by modeling the MMIO peripherals directly. Pretender accomplishes this by recording the original device's MMIO activity, while P2IM instead utilizes blind fuzzing of the entire MMIO layer. These approaches themselves have differing utility; P2IM cannot be used as a generic re-hosting solution, while Pretender requires the original hardware which must be instrumentable. While full automation is an important goal, and we expect that some manual aspects of HALucinator can be automated in the future, HALucinator's HLE approach allows it to handle many cases that neither automated system can. First, both works list DMA as a major limitation; as DMA tends to be used with high-performance peripherals, and its complexity lends itself to being implemented within a library, HALucinator handles DMA by simply removing it from the program. We re-host multiple samples containing DMA in Section 5.3. Second, P2IM only considers sequences of MMIO interactions as input; when a crash is found, this must be mapped back to the external stimulus, requiring a deep understanding of the external peripherals' MMIO interface. HLE-based approaches do not suffer from this problem, as they work only with this external stimulus, and the inputs can be readily replayed against real and virtualized targets alike.

7 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 [24], 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.

8 Conclusion

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`, HALucinator-fuzzer is available at `https://github.com/ucsb-seclab/hal-fuzz`.

Acknowledgments

We thank Bo Feng and Long Lu for providing us with some of the firmware samples used in this paper. We would also like to thank Michael Bailey for his guidance during the editing of this paper.

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 Energys National Nuclear Security Administration under contract DE-NA0003525. This material 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, and by Department of Homeland Security under agreement number FA8750-19-2-0005. 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, Department of Homeland Security, Office of Naval Research, AFRL, DARPA or the U.S. Government. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850868).

References

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

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

[3] Build With Mbed. `https://www.mbed.com/built-with-mbed/`.

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

[5] CSAW Embedded Security Challenge. `https://csaw.engineering.nyu.edu/esc`.

[6] Halucinator: rf233.py. `https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/atmel_asf_v3/rf233.py`.

[7] HALucinator: stm32f4_uart.py. `https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/stm32f4/stm32f4_uart.py`.

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

[9] stm32duino - Arduino_Core_STM32/system. `https://github.com/stm32duino/Arduino_Core_STM32/tree/master/system`.

[10] 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`.

[11] TrustworthyComputing / csaw_esc_2019 - Github. `https://github.com/TrustworthyComputing/csaw_esc_2019`.

[12] P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[13] American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/`.

[14] AFL-Unicorn. `https://github.com/Battelle/afl-unicorn`.

[15] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: A resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):8, 2018.

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

[17] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[18] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, volume 41, page 46, 2005.

[19] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, 2016.

[20] Andrei Costin, Apostolis Zarras, and Aurélien 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*, pages 437–448. ACM, 2016.

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

[22] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2015.

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

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

[25] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki: a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[26] Stack 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.

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

[28] Halvar Flake. Structural comparison of executable objects. In *Proc. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 161–174, 2004.

[29] 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`.

[30] The FreeRTOS Kernel. `https://www.freertos.org/`.

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

[32] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Aurelien Francillon, Davide Balzarotti, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Research in Attacks, Intrusions, and Defenses (RAID '19)*. USENIX Association, 2019.

[33] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. ACM.

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

[35] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. ACM, 2011.

[36] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *WOOT*, 2015.

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

[38] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[39] mbed OS. `https://www.mbed.com/en/development/mbed-os/`.

[40] Stephen McLaughlin, Charalambos Konstantinou, Xueyang Wang, Lucas Davi, Ahmad-Reza Sadeghi, Michail Maniatakos, and Ramesh Karri. The cybersecurity landscape in industrial control systems. *Proceedings of the IEEE*, 104(5):1039–1057, 2016.

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

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

[43] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *BAR 2018, Workshop on Binary Analysis Research*, 2018.

[44] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide 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.

[45] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *IEEE conference on local computer networks*. IEEE, 2006.

[46] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *IEEE Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015.

[47] Jing Qiu, Xiaohong Su, and Peijun Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, 2016.

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

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

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

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

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

[54] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick 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)*, pages 286–297. ACM, 2015.

[55] Dullien Thomas and R Rolf. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de lInformation et des Communications, ser. SSTIC*, volume 5, 2005.

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

[57] Wind River SIMICS. `https://www.windriver.com/products/simics/`.

[58] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[59] ZeroMQ: Distributed Messaging. `http://zeromq.org/`.

A Appendix

A.1 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 5. 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.

A.2 Evaluation of P2IM Firmware Samples

In order to test the applicability of HALucinator to realistic firmware, the authors of P2IM [12] provided us upon request with a portion of their real-world firmware samples. These samples represent multiple CPU manufacturers, and various HAL implementations, as described in Table 6 of the paper.

We 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 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 Section 5.3; we added three additional *Translating* handlers, and the firmware ran without issue.

Finally, the Console firmware uses RIOT OS [17], 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 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 Press samples.

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 6: P2IM case-study firmware sample fuzzing results

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