Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images

Eric Gustafson
edg@cs.ucsb.edu
UC Santa Barbara
Santa Barbara, CA
US

Paul Grosen
pcg@berkeley.edu
UC Berkeley
Berkeley, CA, US

Nilo Redini
nredini@cs.ucsb.edu
UC Santa Barbara
Santa Barbara, CA
US

Saagar Jha
saagar@saagarjha.com
UC Santa Barbara
Santa Barbara, CA
US

Ruoyu Wang
fishw@asu.edu
Arizona State
University
Phoenix, AZ, US

Andrea
Continella
a.continella@utwente.nl
University of Twente
Twente, NL

Kevin Fu
k.fu@northeastern.edu
Northeastern
University
Boston, MA, US

Sara Rampazzi
srampazzi@ufl.edu
University of Florida
Gainesville, FL, US

Christopher
Kruegel
chris@cs.ucsb.edu
UC Santa Barbara
Santa Barbara, CA
US

Giovanni Vigna
vigna@cs.ucsb.edu
UC Santa Barbara
Santa Barbara, CA
USA

ABSTRACT

In today's era of the Internet of Things, we are surrounded by security- and safety-critical, network-connected devices. In parallel with the rise in attacks on such devices, we have also seen an increase in devices that are abandoned, reached the end of their support periods, or will not otherwise receive future security updates. While this issue exists for a wide array of devices, those that use *monolithic firmware*, where the code and data are opaquely intermixed, have traditionally been difficult to examine and protect.

In this paper, we explore the challenges of retrofitting monolithic firmware images with new security measures. First, we outline the steps any analyst must take to retrofit firmware, and show that previous work is missing crucial aspects of the process, which are required for a practical solution. We then automate three of these aspects—locating attacker-controlled input, a safe retrofit injection location, and self-checks preventing modifications—through the use of novel automated program analysis techniques. We assemble these analyses into a system, Shimware, that can simplify and facilitate the process of creating a retrofitted firmware image, once the vulnerability is identified.

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

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

RAID '23, October 16–18, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0765-0/23/10.
https://doi.org/10.1145/3607199.3607217

ACM Reference Format:
Eric Gustafson, Paul Grosen, Nilo Redini, Saagar Jha, Ruoyu Wang, Andrea Continella, Kevin Fu, Sara Rampazzi, Christopher Kruegel, and Giovanni Vigna. 2023. Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607217

1 INTRODUCTION

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

In the last six years, this phenomenon has become tangible to consumers, with the mass-market availability of connected devices for homes and businesses, including thermostats, lighting, physical security, and a variety of sensors. Moreover, these embedded devices also form the backbone of critical infrastructure and drive a wide range of military systems and applications; their security has direct implications for operations both in cyberspace and the real world.

Unfortunately, even when bugs are found and reported, this does not guarantee that a fix will be available. A worrying trend in IoT devices is devices abandoned by vendors [32, 43], or otherwise excluded from support [1], which do not receive security patches. Since embedded systems based on monolithic firmware cannot be simply updated by updating the operating system or libraries, vulnerabilities in these systems have a much longer patch latency due to the extra work involved in creating and testing fixes. For example, the Urgent/11 [33] buffer overflow vulnerabilities affect such a wide variety of real-time operating systems and libraries [11] that it is unclear whether and how a patch will be available for many of these systems. Unfortunately, the increasingly safety- and security-critical nature of many of these devices means that users must patch or replace the device. In some cases, replacing the device may not be physically possible or financially practical, and users must take matters into their own hands.

Since the firmware's source code is generally not available, patching the compiled binary firmware could be the only viable option. Recently, organizations such as the US's Department of Defense have recognized the severity of the problem through the creation of grant programs [14, 35] seeking binary patching technologies. However, in an interesting twist, such solutions are hindered by the security community's own attempts to increase hardware and firmware security. Chip vendors and firmware authors have integrated numerous counter-measures designed to prevent unprivileged access to devices, even with invasive physical access; these include features such as read-protected flash memory on chips and built-in encryption [8, 19, 46]. These were designed to stop malicious attackers, but unfortunately they also reduce the scope of third-party patching capabilities. Even when the firmware can be successfully obtained, numerous challenges still exist that complicate current approaches. Current solutions tackling binary rewriting work primarily on ELF files [27, 49, 51]. These approaches leverage the metadata present in ELF headers to re-arrange code, fixing code offsets and pointers. However, such metadata is often not available for firmware. Many devices that rely on smaller or lower-power CPUs run *monolithic* (or *blob*) firmware, in which the code and data, including libraries, are opaquely intermixed into a single file. To patch it, analyses would need to be able to safely re-arrange code, which requires a complete and accurate control-flow graph—otherwise, unsound approaches would harm the functionality of the device. Thus, in the case of monolithic firmware, we are left with only Detours-style patching [25], where code is inserted into an otherwise-unused region, and the instructions of the program are altered to use it. Worse yet, without metadata, and without an operating system, there is no standard source of input data that a patch can process to make security-related decisions. Finding the ideal location to insert this additional code is also difficult, with no guarantees of available persistent storage, and no simple way of determining content that is safe to overwrite—not to mention space issues due to the limited size of firmware images. Finally, to make these systems robust, firmware typically checks its content to ensure that it is not intentionally or accidentally modified; such checks must be overcome before any kind of binary patching can happen.

Recent work has recognized the importance of updating monolithic firmware with patches [23, 26, 34]. Unfortunately, none of these solutions is complete. HERA [34] and RapidPatch [23] assume the presence of specific run-time environments that are used for patching (a built-in hardware debugging feature on ARM Cortex-M processors and an eBPF-based runtime environment, respectively). Moreover, both systems assume that the firmware includes symbols (so it is easy to find the right place for a patch) and do not worry about the space needed to add code. DisPatch [26] goes one step further and automatically identifies the right patch locations, but only for Robotic Aerial Vehicle (RAV) firmware. It also does not address the questions around the space required for a patch nor the problem of (self-)check routines.

In this work, we take the first steps to enable practical and general end-to-end security retrofitting for monolithic firmware binaries. We first identify the concrete pre-requisites and challenges an analyst needs to consider to perform retrofitting on a given device. Then, we propose novel automated reverse-engineering techniques able to guide the analyst through the process to the maximum extent possible. While the immense hardware and software diversity in monolithic firmware-based devices does not allow for a full automation of the retrofitting process, our techniques automate tedious, time-consuming steps, which let the analyst focus on the task of actually mitigating the vulnerability. Specifically, our techniques perform three fundamental steps that are needed to retrofit firmware: (1) identifying attacker-controlled sources of input, (2) identifying memory locations suitable for inserting a patch, and (3) identifying verification mechanisms that prevent the deployment of a patch.

We combine the aforementioned components into a system, Shimware, that is able to perform all of these tasks on a monolithic firmware image and insert a patch payload to mitigate a given vulnerability. Our system is based on the popular open-source `angr` [45] binary analysis framework, which allows for minimizing effort while the handling of the diverse architectures and binary formats found in firmware. This system enables a technical analyst to quickly retrofit firmware to secure critical embedded industrial, medical, or military systems, when replacement is impossible.

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

In summary, our contributions are as follows:

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

Source code for the analyses in this paper can be found at: https://github.com/ucsb-seclab/shimware

2 BACKGROUND

In this section, we clarify aspects related to firmware retrofitting and describe the challenges that are faced when creating and deploying a patch for a monolithic firmware sample.

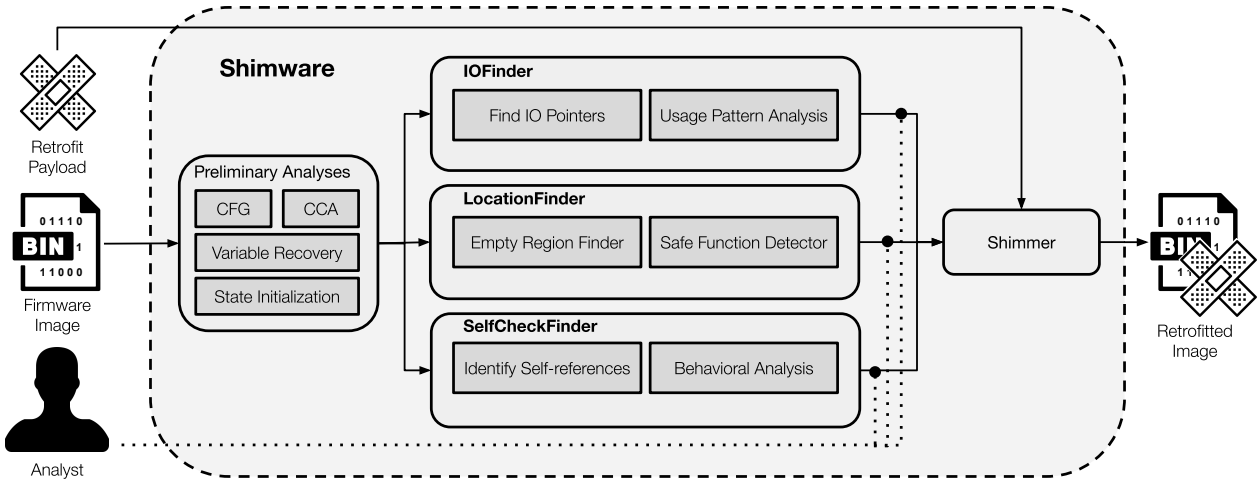

Figure 1: Shimware overview. Our static-symbolic analyses automatically identify attacker controlled sources of input (IOFinder), memory regions where to insert a patch payload (LocationFinder), and self-checks that prevent firmware modifications (SelfCheckFinder). The Shimmer component then assembles a final, modified, firmware image.

Monolithic Firmware. In contrast to more familiar programs that run on general-purpose computers, we refer to the purpose-built, device-specific software running on an embedded system as *firmware*. In particular, many small, low-power, or highly-integrated devices use *monolithic* (also called *bare metal* or *blob*) firmware, which is characterized by an opaque mass of intermixed code and data, with no standardized metadata whatsoever describing its content.

Memory-mapped IO (MMIO). On modern architectures (e.g., modern ARM microcontrollers), when the firmware of a device wishes to access the various internal peripherals of its system-on-a-chip (SoC), it uses memory-mapped IO (MMIO). In this scheme, each peripheral is given a special region in the memory address space; normal load and store instructions targeting this region address the peripheral itself, instead of normal memory. Within each peripheral's region, multiple memory locations (confusingly called *MMIO Registers*) serve different purposes, such as checking the peripheral's status, adjusting its configuration, or sending and receiving data. While these broad classes represent useful archetypes for MMIO registers, there is no standard governing the layout or semantics of these MMIO registers: they are known to vary widely, even between CPU models in the same product line [22]. Understanding these registers requires consulting the chip vendor's datasheet, if available.

2.1 Challenges & Goals

In our threat model, we assume a remote attacker who is able to exploit an arbitrary vulnerability due to a programming mistake. This can be a buffer overflow bug or an error in the program's logic. We assume that the root cause of the vulnerability can be mitigated by introducing or replacing some code (that is, by introducing a patch). The attacker does not have physical access and can only attack the device remotely.

In the following, we walk through the process that an analyst uses to create a patch for an embedded system, and discuss the challenges inherent in each step and our solutions to these problems. At first glance, the patching process would seem to entail simply taking the firmware code, altering some bytes in it to fix the vulnerability, and running the new version. However, the reality of embedded systems—and even the best-practices advocated by the security community—can hinder third-party security retrofitting. Fortunately, some of these steps can be automated by Shimware's analyses, shown in Figure 1.

Obtaining the Firmware Code. For desktop programs, this step is trivial; the analyst already has the program, as they are able to run it. In an embedded system, such as one based on monolithic firmware, this is not so simple. Unfortunately, vendors making their firmware available is incredibly rare, necessitating the extraction of firmware: either from the device itself, or from a mobile or desktop application designed to update it. Embedded systems routinely implement hardware counter-measures (e.g., [9, 46, 48]) to make this challenging for the analyst; bypassing these is an important prerequisite, but is beyond the scope of this work.

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

Inserting a Patch. Once it has been developed, a patch must be inserted into the firmware image. On a system whose firmware

contains a normal filesystem, this could be as simple as replacing an ELF file. As we target monolithic firmware images, we are left with the more difficult challenge of finding a place in the firmware to safely add code, without affecting the original functionality. Unfortunately, we cannot simply insert code next to the source of attacker-controlled data and shift the remainder of the binary, due to the known-hard problem of locating and adjusting all pointers, which would become incorrect due to the shift. Therefore, the most effective option is to insert the additional code in an unused region of the firmware sample, and substitute an instruction near the source of data for a branch to this region [25]. Deciding which regions are safe to use, however, is its own challenge. As a result, our approach, which we call *LocationFinder*, finds either known-unused space on the device's flash memory, or known-expendable code regions.

Deploying a Patch. Most firmware designs include some sort of verification mechanism to verify its integrity, either when the system boots, or when it is upgraded. Which checks we must deal with also depends on our firmware injection vector. We typically have the choice of using either the firmware's own update mechanism to deploy our retrofit or using a hardware injection mechanism such as JTAG or direct flash memory access. If we find an unprotected JTAG port on a device and the firmware's digital signatures are only checked during an over-the-air update, we can bypass this entirely by flashing our own firmware via JTAG. Therefore, in order to successfully deploy a patch, we must mitigate any self-check that affects either our chosen firmware installation vector or the firmware's boot process. We thus propose an analysis, *SelfCheckFinder*, able to identify many forms of self-checks present in firmware by looking for operations utilizing the content of the firmware itself.

In summary, the state of modern firmware protections and hardware countermeasures makes it difficult for an analyst or a tool to patch and deploy a monolithic firmware image. We discuss the conflict between security best-practices and security retrofitting in Section 5. In this work, we aim to aid the analyst in retrofitting firmware, where possible, by automating the tasks of finding sources of attacker-controlled data, safe code injection locations, and code self-checks.

3 METHODOLOGY

In this section, we propose our automated program analysis approach to simplify the process of security retrofitting monolithic embedded firmware. We identify three time-consuming, previously-manual tasks needed for firmware retrofitting, and propose analyses to automate them: locating the firmware's IO routines (in binaries without symbols), finding space for a payload, and mitigating any built-in self-checks on the firmware. The outputs of these analyses can be combined in a final assembly phase to allow an analyst to successfully inject a retrofit payload into a firmware image.

Assumptions. As we have mentioned in previous sections, firmware retrofitting of real devices is by no means a fully-automated process, regardless of method; we make a few assumptions about the scenario our system is used in. First, we assume the analyst is able to extract, and replace, the complete firmware images, including any bootloader stages that perform selfchecks; physically extracting or injecting firmware is out of scope for Shimware. Second, we assume the analyst knows how the vulnerability being patched is triggered, such as via a reproducing input. The analyst must know the architecture and CPU model of the device being patched, either from a datasheet or physical inspection. We assume that the analyst can express the patch merely in terms of inspecting device input; while Shimware's patches are limited only by available space, we discuss how this approach may influence the writing of patches in Section 5.

Workflow example. To outline the process of using the tool, and introduce its components (shown in Figure 1), we will discuss how an analyst would use our system to remediate a security vulnerability in a hypothetical industrial control system, such as the PLC discussed in Section 4. While our system is primarily targeted at facilitating technical analysts in retrofitting firmware, a sufficiently-advanced end user could also leverage the system if needed. In this hypothetical scenario, our analyst works in the public sector, and aims to fix a security vulnerability in an irreplaceable PLC-driven machine, which is being actively exploited over the network.

First, the analyst must gain access to the device, gather some basic facts about it, and obtain its firmware image. Our analyst identifies an unprotected JTAG port on the device, from which they are able to obtain the firmware, and identify the ARM CPU's model number.

Next, the analyst must identify the cause of the vulnerability. In the event of an actively-exploited flaw, this may be readily apparent, but this must be translated into a firmware modification. The analyst notes that a specially-crafted packet is required to trigger the vulnerability, and writes, in C, the payload to detect and stop it. However, the analyst now needs to find the point in the firmware at which new packets are read from the device's network controller. Note that this scenario matches well with prior work on (hot-)patching [23, 26, 34], which assumes that the firmware code and patch are provided.

Using the chip's model number and firmware image, the analyst runs the Shimwareanalyses. This returns the location of a CRC check which needs to be replaced (from SelfCheckFinder), an empty location in the firmware to put the payload (from LocationFinder), and a set of locations where IO is performed in the firmware (from IOFinder), with one labeled as belonging to the onboard Ethernet controller. The analyst provides these details to the Shimmer, along with the payload's source code, and a modified firmware image is produced.

Finally, the analyst uses the previously-discovered JTAG port to deploy the firmware to the device. Naturally, the analyst now needs to ensure that their retrofit did not break any functionality. The analyst uses the PLC's own diagnostic self-tests, as well as diagnostic data for the entire machine, to ensure that it still meets its design specifications with the retrofit in place.

While this process may not be fully-automated, it avoids the need for time-consuming, error-prone, manual reverse-engineering of the firmware. We demonstrate that this process is reasonable by applying Shimwareto real-world devices in Section 4.

All of these analyses are implemented on top of the `angr` program analysis framework. We discuss implementation details related to our usage of `angr` in Appendix A.

3.1 IOFinder

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

As we outline in Section 2, there are two significant complications with locating IO. First, in most modern architectures, particularly ARM, we cannot tell statically which instructions in the binary perform IO operations, as normal load and store instructions are used to access peripherals. Second, the location, layout, and semantics of each hardware peripheral varies widely with the CPU on which the firmware is designed to run. Even when these accesses are located, firmware may perform numerous IO operations that are of no interest to the analyst, such as setting and clearing configuration flags, or checking status registers.

That said, we are able to leverage a few key insights to make this task tractable through automation. First, and most importantly, the location of MMIO-based peripherals is fixed by the hardware and known at compile-time by the firmware's compiler, meaning that the constants representing these peripherals will be observable in the program. However, these pointers can stored in global memory, or used as function arguments to the IO-related functions to select which peripheral to use, leaving us with some indirection to resolve as well. Finally, while the semantics of each peripheral will vary between chip models, we can use metadata shipped with debugging tools to help label these for the analyst, and making their decision of which data to intercept much easier.

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

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

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

Usage Pattern Analysis. Since all of these pointers are usually not declared or accessed near where the actual IO operation occurs (e.g., they are passed into another function as an argument), we utilize symbolic execution to determine how these pointers are used and locate specific IO operations of interest. We chose every function that defines a pointer to an IO memory area, or uses a global memory location that is initialized to an IO pointer, as entry points for symbolic execution. Since many IO operations are inconsequential to the analyst, instead of immediately logging all IO operations encountered, we apply dynamic taint tracking to understand how data is used. During the symbolic execution, we use the following rules:

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

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

External Peripheral Information. The analyst must now decide which peripheral is relevant to their retrofitting scenario, so that its data can be used to develop a retrofit payload. Inevitably, firmware can access data from peripherals that are uninteresting to the analyst, in the same manner as those that represent external input. While these are not false positives in the traditional sense, we can use some information about the microcontroller's hardware to help the analyst quickly locate peripherals related to their retrofitting scenario by labeling the names and registers of peripherals accessed during the symbolic execution step. For example, on the ARM architecture, SystemView Debugger files (SVD) are available in an online repository [36] for many popular embedded ARM CPUs, which can be queried by IOFinder to label the results with names and descriptions.

Listing 2 shows a set of IO-related functions from the `atmel_6lowpan_udp_rx` firmware binary, taken from the dataset used in [12]. This binary implements a 802.15.4 mesh network node, with a radio module controlled over the Atmel SAMR21's SPI bus. During the firmware's boot, the function `trx_spi_init()`

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

Figure 2: Example code from the `atmel_6lowpan_udp_rx` firmware from [12]. Without resolving indirection statically, we cannot see the pointer to the SPI bus being used in `trx_reg_read()`. Names of functions and variables provided only for clarity and not known during analysis.

is called, which sets up the SPI bus selected at compile-time to control the radio and stores a pointer to it in a global struct. Much later, when data is received, the firmware's interrupt handler calls `trx_reg_read()`, to obtain data from the radio. If we were to examine `spi_trx_reg_read()` without its names, and without any other context, we would see a function that adds an offset to a value in memory, and dereferences it, which may or may not be an IO operation, depending on what this global memory value (`&master`) is. When we compute the fully-initialized state, however, we notice `&master->hw` stores a pointer to the SPI bus controller (0x42001800), and this is added to our fully-initialized state. When we run our IOFinder using this state, we notice that `spi_master->SPI.DATA.reg` is a pointer to the IO region and taint the variable `data`. When the value of `data` is returned, the analysis records this as a MMIO read operation, since the data from the SPI bus is being made available to the rest of the program. Since we know that this sample was built for an Atmel ATSAMR21G18A, we can use the available SVD files for this chip to automatically label this access as coming from the `SERCOM4->I2CM_DATA` register, the data register of one of the chip's combined SPI/I2C/USART interfaces.

3.2 LocationFinder

To retrofit a monolithic firmware image, we need to find a location in the binary where we can insert the payload. However, as discussed in Section 2.1, this is not simple. While finding a safe region to insert our payload can be shown to be undecidable – proving that a memory region is not used by a program requires solving the halting problem – we can make some assumptions to find regions in the program that are highly unlikely to be used.

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

Second, we may not always have sufficient free space to insert our payload; firmware binaries typically need to fit into relatively small storage spaces, and are usually compiled with sized-focused optimizations. Without any guarantees on available insertion space, we are left to remove something from the binary itself to make room. This too can be shown to be undecidable; furthermore, existing work on this area [37, 42] relies on having complete, accurate control-flow graph information, which is not possible in this setting.

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

EmptyRegionFinder. This analysis locates regions of the firmware that are unused. We locate contiguous regions of repeating values (typically 0 or 0xFF), and track the largest one found in the binary. We ignore regions that are statically referenced in the binary, such as when pointers referring to the region are used in the program. While we cannot guarantee that a pointer to a nearby location is not used to access a seemingly-empty region, such as in a loop, we note that flash memory, such as found on the devices used in this work, is written page-at-a-time, and is not an efficient storage media for data that needs to be modified often.

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

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

The LocationFinder uses both of these analyses, and picks the largest available region to inject the analyst's payload. The analyst then uses available testing corpora, such as test programs, internal self-checks, and companion apps, to verify that these results are correct.

3.3 SelfCheckFinder

To deploy a retrofitted firmware image, we have to locate any places where the firmware checks its own integrity so that they can be replaced. Defining what a self-check is, however, must be done very carefully. Cryptographic functions are a logical tool for

implementing self-checks, and previous work [21, 29] proposes various static and dynamic approaches to finding cryptographic functions. There are plenty of self-checks (e.g., the simple addition-based checksum) that would not be detected by these schemes, but we would still need to locate them here. Moreover, not all cryptographic functions are self-checks; we only are interested in those which actually involve the content of the firmware. Finally, modern SoCs include hardware support for CRCs and cryptography, meaning that a self-check's actual math operations may not appear in the code at all.

To find self-checks, we make two key observations: First, similar to the IOFinder, our "self-check" must utilize a pointer to the beginning of the region it wishes to check. Theoretically, this is the base address of the binary, but we note that monolithic firmware images that are internally composed of a bootloader and an application may have a scheme in which one region checks another. Therefore, we define a *self-reference* to be a pointer to the firmware, which is page-aligned (on embedded ARM CPUs, aligned to 0x400). Flash memory on embedded CPUs typically only allow writing an entire page at a time, and hence verification of the contents occurs along page boundaries as well.

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

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

Compute the Fully-initialized State. Using the same technique described in Section 3.1, we compute a fully-initialized state, but this time considering only pointers that are self-references.

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

Behavioral Analysis. We employ symbolic execution on the remaining functions, and look for places where the same instruction reads from many locations in the firmware. Precisely, we consider a function a self-check if it accesses at least N locations, where N is also the number of loop iterations allowed during the execution. In short, if for every new iteration, we get an additional access to the firmware, this loop may implement a self-check. We exclude from this set any function whose loop also writes to N locations; these include common primitives for string processing such as `memcpy` or `memmove`.

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

3.4 Shimmer

The final step of Shimwareis to assemble all of the information collected in the previous steps, and the analyst's manually-created payload (the patch), into a modified image that can be deployed to the device. While the content of the analyst's payload is outside the scope of this work, Shimmer allows for payloads to be written in C, after which they are compiled to match the target device's sub-architecture, and size-optimized into a binary blob. The Shimmer module implements a model of retrofitting in which the analyst's payload is triggered right after input is read from the potentially attacker-controlled source (found the the IOFinder). As in Detours [25], the instruction at the trigger location is moved to the start of the patch payload, and replaced with a simple jump instruction pointing at that payload. The only manual task left to the analyst is to choose which source of input to monitor, from the list found in the IOFinder analysis. The Shimmer will replace each detected self-check with a portable checksum routine, and select the largest detected empty region as the payload's injection location. The system supports the application of multiple patches, with the only limitation being a one patch per trigger instruction limit, and the amount of available space within the binary.

4 EVALUATION

To assess the capabilities of our tool, we first perform an evaluation of our IOFinder and LocationFinder on a dataset of samples collected from related work. Then, to evaluate the full sysem end-to-end, we present three case studies where we show how Shimwaresuccessfully led to security retrofitting of three real-world, safety-critical devices. We understand that it would be desirable to expand the evaluation to more than three real-world targets. However, we do want to note the difficulty of extracting firmware from such devices as well as finding and patching relevant vulnerabilities. Unfortunately, this difficulty extends to evaluating the SafeFunctionFinder and SelfCheckFinder subcomponents at any kind of scale. Both require a set of devices for which we have the complete firmware, and a functioning device to properly evaluate, as knowing whether these analyses succeeded requires us to observe the functioning properly; development board samples used in related work do not contain these features for our tools to find[1]. Instead, we evaluate these approaches using the case studies later in this section. The size of our dataset is consistent with prior work on firmware patching: HERA [34] used two vulnerable medical devices and one (existing) vulnerability in the FreeRTOS operating system; RapidPatch [23] was evaluated on bugs in FreeRTOS, ZephyrOS and two libraries (running on five common embedded devices); and DisPatch [26] was applied to two RAV firmware images (3DR IRIS+ and MantisQ).

4.1 IOFinder Evaluation

First, we explore the performance of the IOFinder analysis, on firmware samples obtained from previous work.

We can use development boards and open-source firmware samples, for which source code and symbols are available, to serve as ground truth for the IOFinder analysis. To build our dataset, we obtained 17 samples from related work [12, 20]. We used all available samples that were built as "bare-metal;" we discuss challenges with mbedOS, Arduino, and other library-OS frameworks in Section 5. These samples represent five microcontroller models, from three vendors, with widely-varying peripheral and software driver implementations, and a diverse set of applications, including a PLC, CNC mill, and mesh network nodes. Using these samples,

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

and the hardware for which they were built, we enumerated the set of peripherals that actively communicate with the outside world, as these are the peripherals an analyst would potentially consider as a source of data for a security retrofit. This process was manual, to account for dead code, and to consider only those peripherals that actually transmit and receive data externally. This specifically includes peripherals such as serial ports, buses, and sensors, and excludes timers, clocks, power control, and other common peripherals that do not constitute communication.

Table 1 shows our results; the MCU column indicates which microcontroller model the firmware was designed for, and the "Useful Peripherals" column lists the peripherals determined to be useful for retrofitting. The "Tot. No. Functions" column shows the number of functions in the binary, as identified by `angr`'s static analyses, while "No. Candidate Functions" refers to how many of those were selected for further investigation by our static heuristics. During the dynamic phase of our analysis, the symbolic execution generated many load and store operations from the IO region ("Tot. No. IO Ops" column), of which a much smaller number ("No. Filtered IO Ops.") survived our heuristics and are considered potentially useful to the analyst. Finally, the "No. Useful IO Ops." column shows how many of those operations flagged by the analysis were related to the set of useful peripherals.

With regards to our static analysis phase, the results show that we are able to effectively focus our analysis on the part of the program containing IO, while our computation of the fully-initialized state allows us to draw proper correlations between IO initialization functions and later uses.

During the dynamic phase, the results show that in many cases we are able to significantly reduce the amount of IO the analyst would need to consider. However, we note that the difference between the useful set of peripheral accesses and the set of filtered IO operations reported to the analyst does not constitute false positives in the conventional sense. All of the reported IO operations are indeed valid IO operations, and are useful for various reverse-engineering tasks. Even peripherals that are not important for the task of shimming, such as clocks and timers, can have their data stored and made available to the rest of the program, and would be reported as such. IOFinder automatically applies external labeling information from the `cmsis-svd` [36] database, which labels each peripheral register for all supported CPUs, allowing an analyst to quickly locate any peripheral of interest in the output. Actual false positives could theoretically result when the underlying static analyses are unable to determine the correct calling convention of functions (e.g., `angr` determines a function returns a value to the caller, when it does not), although we did not notice any such cases in the output of this experiment. The analysis does, however, have a few false negatives ("No. Missed IO Ops." column). These cases all stem from an inability to determine a correlation between an IO-related initialization function and an actual IO function, due to the use of nested structs and C++ objects to store the IO configuration. To test this, we implemented a mode for IOFinder where the analyst can manually designate IO-related structs, and we were then able to locate all of the missing peripherals. Future advances in binary type recovery related to structs will help us determine this information automatically.

4.2 LocationFinder Evaluation

As mentioned previously, we require real firmware to assess the ability of LocationFinder to identify "empty" space that can be used to accommodate the code for the patch. Hence, we cannot use ELF files; they do not contain empty space, since this space is created when the device is flashed. Fortunately, we were recently able to obtain the dataset of monolithic firmware from the FirmXRay paper [53]. We randomly selected 50 samples and ran the LocationFinder to get a better understanding of its behavior. It detected an average of 375 bytes of available patching space. This actually excludes two outliers with a massive 65K and 75K of empty space, which would have otherwise significantly increased the average. 375 bytes is more than enough to accommodate most patches (including the ones we created for our case studies). We found space in all 50 samples, although the smallest region consisted of 28 bytes in 3 of the samples. While we cannot completely verify the usability of space without the hardware or ground-truth data, manual analysis confirmed that the LocationFinder is able to find meaningful space in real-world firmware.

4.3 Case Study: Power Supply

We used Shimwareto retrofit an RD DPS5015 [7] lab power supply unit (Figure 3).

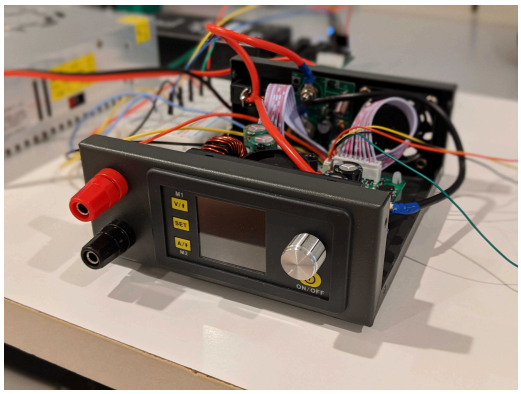

Figure 3: RD DPS5015 power supply, opened [7]

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

Unfortunately, the legitimate functionality of this communication mechanism can allow anyone with network or radio proximity to the device to remove all safety limits, and adjust the voltage or current to any value, causing damage or destruction of the device-under-test, and potentially of the unit itself [18]. While simply disabling network connectivity is one way to make this device safe, we would instead like to add the functionality that the voltage

Table 1: IOFinder Evaluation Results. For each of the 17 samples in our dataset, we report the MCU model, the useful peripherals, and the results of our analysis in terms of reported IO functions and operations. Our filtered IO operations contain all the useful ones.

Sample [Dataset]	MCU Model	Useful Peripherals	Tot. No. Functions.	No. Candidate Functions	Tot. No. IO Ops.	No. Filtered IO Ops.	No. Useful IO Ops.	No. Missed IO Ops.	Actual Useful Peripherals
atmel_6lowpan_udp_tx [12]	ATSAMR21G18A	ETH(SPI),I2C,UART	533	182	91	46	21	2†	ETH(SPI),I2C
atmel_6lowpan_udp_rx [12]	ATSAMR21G18A	ETH(SPI),I2C,UART	533	182	91	46	21	2†	ETH(SPI),I2C
p2im_cnc [20]	STM32F429	UART	331	121	110	32	3	0	UART
p2im_drone [20]	STM32F103	UART,I2C	230	71	39	38	18	0	UART,I2C
p2im_robot [20]	STM32F103	I2C,UART	205	41	59	22	15	0	I2C,UART
p2im_soldering_iron [20]	STM32F103	DMA(ADC),I2C(IMU),I2C(OLED)	371	99	107	40	19	4†	DMA(ADC),I2C(IMU)
samr21_http [12]	ATSAMR21G18A	UART,ETH(SPI)	324	68	61	26	8	0	UART,ETH(SPI)
samr21_uart_polling [12]	ATSAMR21G18A	UART	44	35	28	16	3	0	UART
samr21_fatfs_usd [12]	ATSAMR21G18A	SPI(SDIO),UART	207	189	51	25	13	0	SPI(SDIO),UART
st-plc [12]	STM32F401	UART(Wifi),SPI,ADC	981	278	142	49	11	0	UART(Wifi),SPI,ADC
stm32_tcp_echo_client [12]	STM32F469	ETH,I2C	477	63	86	25	19	0	ETH,I2C
stm32_tcp_echo_server [12]	STM32F469	ETH,I2C	478	62	80	26	19	0	ETH,I2C
stm32_udp_echo_client [12]	STM32F469	ETH,I2C	468	58	86	25	19	0	ETH,I2C
stm32_udp_echo_server [12]	STM32F469	ETH,I2C	463	58	80	25	19	0	ETH,I2C
nxp_uart_polling [12]	MK64F12	UART	108	33	36	11	4	0	UART
stm32_fatfs_usd [12]	STM32F469	I2C,SDIO	276	42	81	33	26	0	I2C,SDIO
nxp_fatfs_usd [12]	MK64F12	SDHC,UART	240	59	64	17	6	0	SDHC,UART

†: We managed to cover these false negatives by providing our system with additional knowledge about the employed data structures (Section 4.1).

limits specified by the operator on the physical device represent a maximum of what the remote automation can set.

We obtained the popular OpenDPS firmware used with this device [4] by installing it onto our unit, and then dumping it via the device's exposed SWD debugging port. This yielded a monolithic firmware image, in which `angr`'s CFG recovery detected 420 functions. The IOFinder analysis detected functions containing IO references, out of which 109 actually performed interesting IO when executed. Among these was a clearly-labeled access to the USART1 data register (USART1->DR), which would serve as a source of input data. LocationFinder discovered a region of 872 bytes between what appears to be the bootloader and the primary application of the firmware. The SelfCheckFinder located and replaced exactly one self-check: the use of CRC16 to validate the firmware image.

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

4.4 Case Study: PLC

We employed Shimware to retrofit an Allen-Bradley ControlLogix 1756 PLC [44]. This high-end, but end-of-life, product is suitable for large automation tasks, including factories and military applications. The system comes in the form of a chassis with at least one CPU card, and numerous IO devices depending on the application. Our configuration contains the L64 CPU card, an analog output card, and an Ethernet card. The CPU contains a custom ARM7TDMI-S derivative CPU, with numerous custom ASIC components controlling network and communications features, presenting a unique challenge for our approach. While we do not know the MMIO layout of the main CPU, we know we are looking for input from the chassis backplane, a proprietary high-speed bus, which is therefore likely to be using some form of DMA.

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

We obtained the 2MB firmware image through the manufacturer's website. CFG recovery yielded 8,937 functions in the binary. Of these, 593 had IO pointers. When executed, IOFinder detected 340 unique IO operations. Since the CPU is custom, we had to perform the additional step of reverse-engineering the CPU's MMIO layout, but were helped by our analyses in doing so, as we could focus our efforts on those portions of the code detected by IOFinder. Since the amount of detected IO locations is numerous, and we estimate that this result is legitimate, we focused on those locations which performed DMA-based operations. This only consisted of 31 IO-related instructions, representing five unique MMIO registers. All of these appeared to be related to a peripheral at 0x40000000, which turned out to be the DMA controller for the backplane.

SelfCheckFinder detected two self-checks, a checksum and CRC, calculated on the whole firmware image; the Shimmer replaced these both with a checksum, allowing the firmware to boot.

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

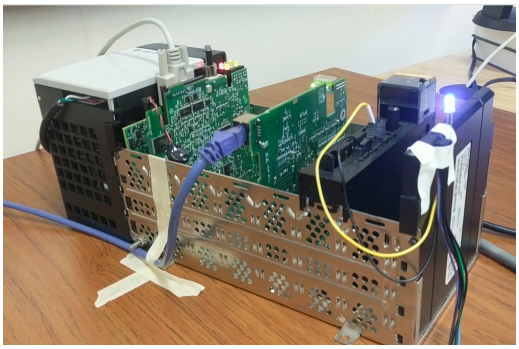

Figure 4: AB ControlLogix PLC [44], disassembled. The LED's blink is clamped in the firmware by our retrofit payload.

Using the above information provided by our analyses, we injected a payload into the firmware, which is able to filter incoming Common Industrial Protocol (CIP) messages, to ensure that no message alters the variables of a running ladder-logic program outside of compiled-in parameters. We tested the system for correct behavior by running a test ladder-logic program on the device, and by connecting the RSLogix companion software to monitor the device's behavior. Our ladder logic program causes an LED connected to one of the PLC's output cards to blink at a certain rate (see Figure 4); we used our payload to clamp these values between 2 and 10 Hz. We verified that, if a CIP message is received that would set this rate outside the bounds, that it is ignored, and a message is sent to the operator. We should also note that, like many safety-critical devices, this PLC has numerous self-tests, watchdog timers, and hardware safeguards, both at boot-time and continuously at run-time, which would have alerted us to any faults due to our retrofit.

4.5 Case Study: Pacemaker Monitor

We used Shimwarewith a Medtronic MyCareLink 25000 pacemaker monitor. This small hand-held device acts as a bridge between the patient's phone (via Bluetooth) and a pacemaker (via short-range radio). The patient is instructed by the device's companion app to use the device daily to transmit cardiac data to their doctor.

Unfortunately, we (manually) discovered a flaw in how the device employs cryptography to keep unauthorized apps and devices from connecting to it. This allows any attacker in range of the device to connect to it, and issue a broader set of commands to the pacemaker than the manufacturer intended, along with accessing the contained medical data. For ethical reasons, we will not speculate on any effects this may have on the implant itself. We reported this flaw to the vendor, which designated it as the high-severity CVE-2020-25183. While we did work with the vendor on an official patch, the root cause here is one we can fix via a retrofit, namely stopping the original cryptographic bruteforce attack. The device contains an STM32F103 Cortex-M3 CPU, and accesses its Bluetooth controller via a serial port. An unprotected JTAG port is present, via a pogo pin connector.

We dumped the full 1MB firmware image, including its bootloader, from the JTAG port. CFG recovery yielded 3,687 total functions. IOFinder yielded 696 candidate IO functions. During dynamic analysis, this produced 93 potentially-useful IO operations, which were automatically labeled with their peripheral names. Among these was the USART3 peripheral operating in DMA mode, our desired source of IO. LocationFinder found a large (250k) unreferenced region of flash memory at 0x80bf96c, between the firmware itself and the non-volatile storage portion of the flash memory. SelfCheckFinder located five self-checks, three of which were false positives, and two of which were hardware-backed CRC32 of the entire firmware, performed at boot-time. These false positives were related to the firmware upgrade routine itself, which naturally loops over the firmware and manipulates the flash memory controller. While this may seem problematic, we note that the firmware and app still work with this feature removed, which also has the positive side-effect of preventing a potential conflict between our patch and the manufacturer's.

We tested the LocationFinder and SelfCheckFinder's results by inserting our retrofit payload into the firmware, and deploying it over JTAG. When the serial port connected to the Bluetooth chip is written to, we check the outgoing payload against the error message sent when an incorrect cryptographic key is used. If this message is found, we reset the device, causing a significant delay, and dramatically lowering the chance of a successful attack. Because this device is only intended to be powered on for a few minutes at a time, this will sufficiently mitigate the risks.

The device's companion Android app has exactly one function: downloading the implant's data and sending it to a doctor. It has no menus or configuration, and after the various steps (connecting via Bluetooth, connecting to the implant, and downloading the data), will display a large green checkmark, indicating that it succeeded. Our validation for this retrofit, therefore, involved ensuring that this process still succeeded with our payload in place. We flashed our modified firmware on the device, and installed the latest version of the vendor's app on an Android phone. We then obtained a compatible, non-implanted, pacemaker, used the app and device to download its data, and ensure that the app reported a successful transmission.

5 DISCUSSION

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

IOFinder. Our IOFinder currently does not work effectively on firmware created on top of large firmware frameworks, such as Arduino [2] or ARM's mbed [30], which utilize a high level of abstraction, and object-oriented programming techniques to simplify development, and ease porting to many hardware platforms. As these requirements do not exist in final commercial devices, vendors instead opt to use lighter-weight packages, such as those provided by the compiler vendor, or creating their own. A careful reader may note that these large frameworks present the ideal conditions for a library-matching approach [12, 16], as the additional, standardized code in these packages is simpler to locate by matching. Therefore, these two approaches are directly complimentary; library matching works well on larger libraries where static analysis fails, but IOFinder works in situations where the libraries to match cannot be obtained, or are customized.

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

Shimmer. Our system's approach to patching relies on inspecting the source of a device's input, akin to a traditional network intrusion detection system. This gives it a great degree of flexibility, and removes a lot of the need to understand the firmware's internals. However, this approach does make some kinds of patches easier to express than others. Inspecting complex protocols, for example, would require one to embed enough parsing in the patch to detect the exploit, which could run into size constraints. We still believe this is far preferable to reverse-engineering the entire firmware image, however.

As with any software patch, once attackers know of its existence, there is some possibility the attacker will be able to evade a naively-written patch, by, for example, crafting equivalent inputs that evade filtration. We note that Shimmer's C-based language allows patch authors to write more robust patches, but again there is a size trade-off to consider.

The Future of Security Retrofitting. As we discuss in Section 2, many factors beyond the control of the analyst can influence the patchability of a system. Interestingly, many of these mechanisms, such as hardware roots-of-trust, software signature verification, and the disabling of hardware code flashing mechanisms, are often rationalized as ways of increasing the security of the code against unauthorized modification. Indeed, the increasingly common hardware-backed integrity verification methods do cleanly accomplish this goal, while leaving some possibility of patching. For example, many boot-time verification schemes in larger systems (e.g., PC SecureBoot and mobile phone bootloaders [40]) allow for the possibility for the user to "unlock" this chain of trust, and run their own code, at their own risk, if desired.

It is very important, however, to distinguish these techniques from those that instead aim to implement intellectual property protection, such as flash read-back protection [31, 46], which are commonly found in some form in most embedded microcontrollers. While these are also commonly touted as security measures, they only achieve this goal when paired with one of the above verification approaches. Unfortunately, for those vendors who indeed wish to add intellectual property protection measures, these methods are directly at odds with security retrofitting, as the firmware itself cannot be easily obtained.

Unless something is done to strike a balance between security, IP protection, and the end-user's ability to repair their own devices, the number of unpatched, abandoned devices will continue to increase. One solution that still allows for both kinds of protection is to implement a means of relinquishing control of these protections when the device's period of support expires. In the case of IP protection, this aligns with the fact that the vendor has no more commercial interest in the product, and, therefore, has no need to obfuscate their firmware further. These ideas seem a natural fit for recent and future IoT security-related legislation [3, 5] being proposed worldwide, intended to help inform users about the support and security policies of their devices. Indeed, the European proposal for the Cyber Resilience Act [13] includes an objective for ensuring that manufacturers improve the security of products throughout their whole life cycle.

6 RELATED WORK

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

Wenzl et al. [54] dissect this area of research, and define four common steps to binary rewriting: *Parsing*, *Analysis*, *Transformation*, and *Code Generation*. The work in this area varies in terms of how they handle each of these steps.

The largest class of this work concerns *reassemblable disassembly* [49–51], the notion of disassembling a binary completely into standard assembly code for the architecture in question, adjusting the code, and then simply assembling it again into the finished binary. This is a preferable technique for the analyst, as it makes the Transformation and Code Generation step easy; assembly can be easily patched manually or automatically via the human-readable assembly code, and turned into a binary again with standard tools.

However, this work presents problems when it comes to tackling the rewriting of firmware. First, reassembly assumes that the firmware can be disassembled completely during the Parsing phase. This includes the inter-related problems of distinguishing code and data, finding function boundaries, distinguishing pointers from integers, and the resolution of indirect jumps. Unless we can re-host the firmware into an emulated environment, we are left with performing those tasks entirely statically. We also are dealing with monolithic firmware images, and therefore do not have an explicit memory map, symbols, or other metadata that can make these steps easier. Therefore, we cannot guarantee the completeness of the disassembly, control-flow recovery, or function identification. As an indicator of this challenge, the samples in Section 4 have an average of 401 unresolved indirect jumps after `angr`'s resolution mechanisms (based on [42]). Since monolithic firmware is not position independent by nature, if we cannot locate every pointer, including those used to access code and data, and adjust them during reassembly, the firmware will not execute correctly.

On top of this, many aspects of these approaches are architecture-specific, with most of works focusing only on Intel x86 binaries [10, 15, 49, 50], or requiring x86 hardware extensions [52]. Some caveats related to ARM instruction set features that inhibit the Parsing and Analysis phases were addressed by Kim *et al.* [27], but the approach still inherits the aforementioned severe limitations of reassembly.

Another emerging class of work revolves around transferring patches from one program to another. OSSPatch [17] works by leveraging source code of open-source projects used in the unpatched target binary to retrofit patches from their upstream sources. A similar work [24] aims to transfer the patched portions of compiled binaries, without the source code. In our scenario, the patch has simply not been developed, and will not be developed by the manufacturer, therefore we cannot leverage either of these approaches.

Finally, while the above work has explored the Parsing, Analysis, and Transformation aspects of binary rewriting, little thought has thus far been given to the Code Generation portion. Unfortunately, this is where the unique challenges of embedded systems, and particularly safety-hardened systems, begin to restrict our ability to patch. First, all of the above work assumes that, when a patched binary is created, the system will simply execute this binary instead of the original. As we discuss in Section 2, most commercial embedded systems contain self-checks to prevent modification or corruption of the firmware (e.g., CRCs and checksums), or intentional manipulation by an attacker or user (e.g., digital signatures) which must be located, and bypassed, for a patch to work. On top of this, it is also assumed by reassembly-based approaches that we have the toolchain needed to create a functioning binary image. In the case of a monolithic firmware image, we do not know what format, if any, is present in the firmware, or what tools may have been used to create it.

Additionally, we need to find space for our added code, either within the image, or by appending it to the end. Since we often obtain monolithic firmware in the form of full flash images (e.g., we cannot simply append data to the end), we must decide what in this image can be removed, or what apparently-empty space is available. This relates to the problem of *binary debloating*, the known-undecidable problem of removing unnecessary content from a program. Recent work has proposed solutions for debloating Dockerized applications [39], during program compilation [38], or using shared library files [6]. Naturally, these kinds of approaches involve extensive knowledge of the program, in the form of source code, object code, or simply do not apply to the firmware domain.

Razor [37] uses a series of test-cases, along with the collection of execution traces and heuristics to determine removable portions of code. Unfortunately, as we mention in Section 2, this kind of trace collection is not possible in the embedded systems domain, without a successful re-hosting solution. Redini et al. [42] propose BinTrimmer, which aims to remove unnecessary code, without using any of the above assumptions. It does so through improvements to indirect jump resolution, which are used as the basis for `angr`'s own indirect jump resolver, used heavily in this work. Unfortunately, its strict dependence on the ability to resolve *all* indirect jumps makes this approach inapplicable to our domain.

In summary, these numerous issues combine to rule out any *Dynamic* or *Full Translation* approaches (as defined by Wenzl et al. [54]), and we must resort to the more simplistic *Direct* or *Minimal Invasive* techniques. Even these (such as Detours [25]) require us to solve at least the problems of code insertion, self-checks, and sources of security-relevant data, which we focus on in this work.

A recent and related line of work has focused specifically on the patching of firmware images [23, 26, 34]. While the authors share our goal and make a convincing case for the need to patch firmware, there are several important differences. Most techniques [23, 34] assume that the location of the patch is known, and that space and self-checks are not concerns. Instead, the focus of this prior work is on hot-patching and ensuring that the patched code can be deployed at run-time. This is in contrast to Shimware, which modifies and installs a new version of the firmware, but which is explicitly designed to help address the real-world challenges of proper code insertion. While we did not explore HERA-style hot-patching in this work, as some of the targets do not support its hardware requirements, the idea of hot-patching and Shimwareare orthogonal, and could likely be easily combined. Dispatch [26] is closest to our work in the sense that the authors attempt to automatically locate the location (function) where the patch should be applied (similar in spirit to our IOFinder). However, their approach relies on the implementations details of Robotic Aerial Vehicle (RAV) firmware to function. It also does not address the questions around the space required for a patch nor the problem of (self-)check routines.

7 CONCLUSION

In this work, we explored the challenges and solutions to security retrofitting of monolithic embedded firmware. We identified the tasks of locating the attacker controlled IO, finding safe payload injection locations, and mitigating self-check functions as candidates for automation. To this end, we proposed three novel static-symbolic analyses that locate these features in a firmware image automatically, and significantly reduce the analyst effort. Our prototype system, Shimware, combines these techniques with a tool that assists analysts in injecting patch payload into the firmware without the need of performing complex reverse engineering. We employed the full system to address three safety and security-critical vulnerabilities in off-the-shelf products from the engineering, healthcare, and industrial automation sectors. Our results show the promise of security retrofitting, even in the challenging context of monolithic firmware, and we hope that future advances in securing embedded firmware will continue to allow users and analysts alike to secure their devices, when manufacturers may not.

ACKNOWLEDGMENTS

This material is based upon work supported by the Office of Naval Research (ONR) under Award No. N00014-20-1-2632, by a gift from Meta, by the INTERSECT project (Grant No. NWA 1160.18.301) funded by the Netherlands Organisation for Scientific Research (NWO), and by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project 'FirmPatch'. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of ONR, NWO, EZK, or Meta.

REFERENCES

[1] 2019. D-Link Adds More Buggy Router Models to 'Won't Fix' List. https://threatpost.com/d-link-wont-fix-router-bugs/150438/.
[2] 2020. Arduino. http://arduino.cc/.
[3] 2020. CA S.B. 327. https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180SB327.
[4] 2020. OpenDPS. https://github.com/kanflo/opendps.
[5] 2020. S.734 - Internet of Things Cybersecurity Improvement Act of 2019. https://www.congress.gov/bill/116th-congress/senate-bill/734.
[6] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Annual Computer Security Applications Conference*. 70–83.
[7] AliExpress. 2020. RD DPS5015. https://www.aliexpress.com/item/32702714880.html.
[8] Amazon. 2020. Creating a code-signing certificate for the Texas Instruments CC3220SF-LAUNCHXL. https://docs.aws.amazon.com/freertos/latest/userguide/ota-code-sign-cert-ti.html.
[9] Frank Armstrong. 2013. A Discussion on Atmel Lock Byte and Firmware Protection. https://www.avrfreaks.net/sites/default/files/A%20discussion%20on%20Atmel%20Lock%20Bits.pdf.
[10] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.

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

[12] Clements, Abraham and Gustafson, Eric and Scharnowski, Tobias and Grosen, Paul and Fritz, David and Kruegel, Christopher and Vigna, Giovanni and Bagchi, Saurabh and Payer, Mathias. 2020. HALucinator: Firmware Re-hosting through Abstraction Layer Emulation. In *USENIX Security Symposium*.

[13] European Commission. 2022. Cyber Resilience Act. https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act.

[14] DARPA. 2020. Assured Micro-Patching (AMP). https://www.darpa.mil/program/assured-micropatching.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Security and Privacy*.

[16] Steven H.H. Ding, Benjamin C.M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *IEEE Security and Privacy*.

[17] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.. In *NDSS*.

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

[19] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.

[20] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*.

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

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

[23] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *Usenix Security Symposium*.

[24] Y. Hu, Y. Zhang, and D. Gu. 2019. Automatically Patching Vulnerabilities of Binary Programs via Code Transfer From Correct Versions. *IEEE Access* 7 (2019), 28170–28184. https://doi.org/10.1109/ACCESS.2019.2901951

[25] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *USENIX Windows NT Symposium*.

[26] Taegyu Kim, Aolin Ding, Sriharsha Etigowni, Pengfei Sun, Jizhou Chen, Luis Garcia, Saman Zonouz, Dongyan Xu, and Dave Tian. 2021. Reverse Engineering and Retrofitting Robotic Aerial Vehicle Control Firmware using DisPatch. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*.

[27] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Annual Computer Security Applications Conference (ACSAC)*. 412–424.

[28] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*.

[29] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *ACM Symposium on Information, Computer and Communications Security (CCS)*.

[30] mbed OS 2020. mbed OS. https://www.mbed.com/en/development/mbed-os/.

[31] Microchip. 2020. AT16743: SAM V7/E7/S7 Safe and Secure Bootloader. http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42725-Safe-and-Secure-Bootloader-for-SAM-V7-E7-S7-MCUs_AT16743_ApplicationNote.pdf.

[32] Jose Nazario. 2017. The problem with patching in addressing IoT vulnerabilities. https://www.fastly.com/blog/problem-patching-addressing-iot-vulnerabilities.

[33] Lily Hay Newman. 2019. Decades-Old Code Is Putting Millions of Critical Devices at Risk. https://www.wired.com/story/urgent-11-ipnet-vulnerable-devices/.

[34] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *Symposium on Network and Distrbuted Systems Security (NDSS)*.

[35] Office of Naval Research (ONR). 2020. Total Platform Cyber Protection (TPCP). https://www.onr.navy.mil/-/media/Files/Funding-Announcements/BAA/2017/N00014-17-S-B010.ashx.

[36] Osbourne, Paul. [n. d.]. CMSIS-SVD Repository and Parsers. https://github.com/posborne/cmsis-svd.

[37] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium*.

[38] Anh Quach and Aravind Prakash. 2019. Bloat Factors and Binary Specialization. In *ACM Workshop on Forming an Ecosystem Around Software Transformation*.

[39] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Joint Meeting on Foundations of Software Engineering*. 476–486.

[40] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. BootStomp: On the Security of Bootloaders in Mobile Devices. In *USENIX Security Symposium*. Vancouver, BC.

[41] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *IEEE Security & Privacy*.

[42] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 482–501.

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

[44] Rockwell Automation. 2020. ControlLogix and GuardLogix Controllers. https://literature.rockwellautomation.com/idc/groups/literature/documents/td/1756-td001_-en-p.pdf.

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

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

[47] subwire. 2020. Autoblob: Automatic Blob-loading for CLE. https://github.com/subwire/autoblob.

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

[49] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[50] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling.. In *USENIX Security Symposium*. 627–642.

[51] S. Wang, P. Wang, and D. Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

[52] Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. 2017. Binary Code Retrofitting and Hardening Using SGX. In *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[53] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In *ACM Conference on Computer and Communications Security (CCS)*.

[54] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique - A Survey on Binary Rewriting. In *ACM Computing Surveys (CSUR)*.

A APPENDIX

We implemented Shimwarein Python using the `angr`[45] binary analysis framework. This allowed us to easily implement both static and dynamic symbolic analyses in the same framework. `angr` also offers broad architecture support, and the flexibility to handle monolithic firmware images.

Loading the Binary. In order to perform analyses on any monolithic firmware image, we first have to obtain the base address, entry point, and architecture of the binary. Doing this automatically is a known open problem, but we leveraged an `angr` plugin that automates this process for many ARM-based firmware images [47], including all of the binaries used in this work. This produces a representation of the device's memory with the loaded firmware, but before any execution begins.

Initial Static Analyses. Once the binary is loaded into simulated memory, we use `angr`'s built-in `CFGFast` analysis, which first uses prologue scanning to locate possible functions within the firmware,

and computes a static control-flow graph starting from these functions. It then attempts to recover the calling conventions of each function, recover program variables (similar to [28]), and computes cross-references to functions and data.

Fully-Initialized State. Along with the above initial processing provided by `angr`, we created a new analysis primitive, used by all the subsequent analyses, which attempts to statically reconstruct the execution state of the firmware, as if it had completed its initialization. This becomes critical when firmware uses global variables, populated at boot-time, to hold pointers to its IO devices, or other data needed for successful execution during any of our analyses. However, to avoid over-constraining the program during symbolic execution, this analysis specifically targets the initialization of pointers.

At a high level, this analysis tracks constant pointers within the program, from where they are computed or loaded from the firmware, to where they are stored to statically-known regions of RAM. To do this, we start from the entry point of the program, and perform a conventional constant propagation on each function, following the topological order of the program's callgraph. Any value which is propagated into memory, which is also known to be a constant pointer, is reflected in the fully-initialized state. For the purposes of this analysis, we define a constant pointer as any integer that could represent an address in the architecturally-defined RAM or IO regions. These constant pointers are also propagated into function calls; a function is only propagated after all of its callers, and will be propagated multiple times, one for each set of arguments used at a callsite. Any conflicting memory writes will result in an unconstrained symbolic value being stored in the given location.

With the initialized state computed, we can then make use of this data in the static and dynamic phases of our analyses. We can now find memory locations that were initialized to hold pointers to MMIO, and use these to find the otherwise-invisible IO functions that use them. Similarly, we use the same technique to find self-references for SelfCheckFinder.

Symbolic Execution. Our analyses use under-constrained symbolic execution to examine a function's behavior, by starting at each function's first instruction, and using the fully-initialized state. To make this tractable on large binaries, we used a number of `angr`'s Exploration Techniques to limit how much execution we perform. We limited the number of basic blocks in any given path to 10,000, the amount of time spent on a function to 5 minutes, as well as using depth-first search and removal of dead-ended paths to reduce memory usage. For IOFinder, we also used the taint tracking and adaptive inter-function level mechanism proposed in previous work [41], to focus our analysis on relevant portions of code.