IRQDebloat: Reducing Driver Attack Surface in Embedded Devices

Zhenghao Hu
New York University
huzh@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

***Abstract*—Embedded and IoT devices often come with a wide range of hardware functionality, but any particular end user may only use some small subset of these features. However, even unused hardware features are accompanied by potentially buggy driver code, which increases the attack surface of the device. In this paper, we introduce IRQDebloat, a system for disabling unwanted hardware features through automated firmware rewriting. Building on the insight that external inputs to the system are typically delivered through interrupt requests (IRQs), IRQDebloat systematically explores the interrupt handling code in the target firmware, identifies the handler function for each peripheral, and finally rewrites target firmware to disable the handlers that correspond to undesired hardware features. In our experiments we demonstrate IRQDebloat's effectiveness and generality by identifying IRQ handlers across four different operating systems (Linux, FreeBSD, VxWorks, and RiscOS) and seven different embedded platforms, and disabling selected peripherals on real-world hardware (a Raspberry Pi and a Valve Steam Link). On the Steam Link, we survey the attack surface and find that disabling selected peripherals could block up to 44 CVEs found in the Linux kernel over the past five years.**

I. Introduction

Many embedded and IoT devices offer a wide range of functionality to support the needs of a diverse user population. However, this breadth of features comes at a price: each feature requires additional software support, and creates more opportunities for vulnerabilities. For hardware features, these vulnerabilities often lurk in kernel-mode driver code, where bugs can lead to complete compromise of the system.

For example, Cisco Meraki WiFi access points integrate a Bluetooth Low Energy (BLE) beacon to provide services such as indoor localization [1]. Although many customers may not need or desire these features, the software needed to support it is enabled in the firmware and remains a part of the device's exposed attack surface. Indeed, researchers from Armis discovered multiple remote code execution vulnerabilities (collectively dubbed "BleedingBit") in the BLE stack of these devices [42].

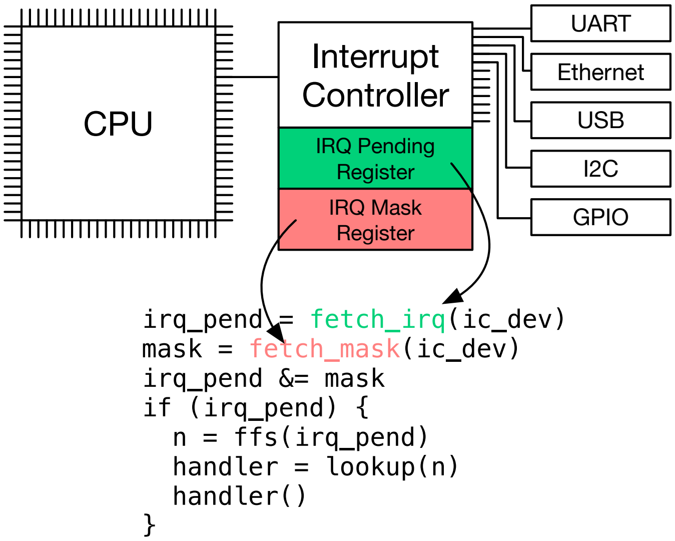
Examples like these demonstrate the need for *debloating*: the automated removal of unwanted features in software. Although debloating has been previously studied in the context of desktop [38], server [31], mobile [47], [27], and web applications [8], less attention has been paid to disabling potentially vulnerable *hardware* features in embedded devices.

In this paper, we demonstrate an approach to debloating that allows users to selectively disable unwanted hardware features by automatically rewriting their firmware. Our key insight is that the the vulnerable attack surface from driver code can, in most cases, only be reached when input from the outside world enters the system via a hardware interrupt. Thus, by enumerating the interrupt handlers on a system, matching them to actual hardware functionality, and then rewriting the firmware to disable code that handles interrupts from unneeded devices, we can effectively close off the driver attack surface from the outside world.

We envision that this capability will be most useful to technically sophisticated users who wish to deploy embedded devices without such unwanted hardware functionality that may render them less secure. A classic (though perhaps apocryphal) example is the anecdote that some secure government facilities may have filled the USB ports of their computers with epoxy to prevent the use of USB [24]; blocking hardware features through firmware rewriting is similar in spirit to this technique, but requires less manual effort (and is more reversible). More prosaically, a large company deploying a fleet of wireless routers such as the Meraki access points mentioned above might wish to disable BLE functionality entirely. And finally, hobbyist users may wish to keep their embedded devices running past their end-of-life date by disabling peripherals whose drivers have unfixed vulnerabilities.

To demonstrate this idea, we have built a prototype firmware debloating system, IRQDebloat. Starting from a snapshot of the CPU and memory state from a real embedded device, IRQDebloat migrates this snapshot into a software emulator and collects execution traces by systematically exploring different paths through a top-level interrupt handler. Once these traces are collected, we use differential slicing [29] to precisely identify the handlers available for each peripheral in the system. Finally, we instrument these handlers on the real device and replace the interrupt handler for that peripheral with one that simply ignores the interrupt and returns.

We evaluate our system on two CPU architectures (ARM and MIPS), four different operating systems (Linux, FreeBSD, VxWorks, and RiscOS) and across seven different embedded system-on-chip (SoC) platforms and find that it can successfully enumerate and identify all registered interrupt handlers. We also demonstrate, in a case study, the use of IRQDebloat to protect a real-world device, the Valve Steam Link, by automatically reverse engineering its interrupt sources, locating the interrupt used for WiFi and Bluetooth, and then disabling it. We estimate that this would prevent 13 distinct CVEs found in the Linux Kernel's Bluetooth and WiFi drivers over the past five years. IRQDebloat allows users to effectively reduce their exposure to unknown vulnerabilities by automatically removing unwanted hardware functionality.



```
irq_pend = fetch_irq(ic_dev)
mask = fetch_mask(ic_dev)
irq_pend &= mask
if (irq_pend) {
  n = ffs(irq_pend)
  handler = lookup(n)
  handler()
}
```

Fig. 1. Typical flow for interrupt handling in an embedded system.

II. Background

A. Interrupts

In an embedded system, most inputs from the outside world are delivered via *interrupts* (sometimes called *interrupt requests* or *IRQs*). Interrupts originate in a hardware peripheral and cause an asynchronous control flow transfer to an *interrupt handler*, a function that services the interrupt. The interrupt handler then communicates with the peripheral that raised the interrupt, handles any pending I/O for the peripheral,[1] and then acknowledges the interrupt, marking it as finished.

In hardware, an embedded CPU typically has only a small number of dedicated interrupt pins (e.g., on ARM, there are only two top-level interrupt pins, one for standard interrupts and another for low-latency "fast interrupts", known as FIQs). To support a larger number of interrupts, most systems come with a dedicated *interrupt controller (IC)*, which multiplexes multiple interrupt pins into a single interrupt source. Interrupt controllers can also be chained together to support an arbitrary number of peripherals by tying the output pin of a secondary interrupt controller to one of the input pins of the primary interrupt controller.

A typical example of IRQ handling is depicted in Figure 1. First, the peripheral signals an interrupt; this causes the interrupt controller to store the interrupt number as a bit in a bitset in a memory-mapped register and then signal the main CPU. The CPU responds to the interrupt by jumping to an architecture-defined top-level interrupt handler. This handler then retrieves the interrupt number from the interrupt controller, and uses it to dispatch to the handler for the specific peripheral. This peripheral-specific handler then handles I/O for the peripheral using memory-mapped I/O (MMIO), and finally acknowledges the interrupt. The details of this process may differ depending on the specific model of interrupt controller; for example, the interrupt controller used by the BeagleBone stores the pending interrupt number as an 8-bit value rather than a bit in a bitset.

To service interrupts, the embedded operating system must maintain a mapping between hardware IRQ numbers and the corresponding handlers. In the simplest case, these could be hardcoded (e.g., using an array of function pointers to the individual peripheral handlers), but more sophisticated operating systems will typically support dynamic registration of IRQ handlers. For example, in Linux, drivers can register a handler by calling `request_irq`, and the mapping between IRQ numbers and handlers is maintained in a radix tree.

Although the implementation details for interrupt handling differ between embedded architectures, the core pattern of receiving a top-level interrupt and then dispatching to a specific interrupt handler appears to be universal. We leverage this pattern to build IRQDebloat, which enumerates the individual peripheral handlers and allows the user to disable them, closing off that peripheral's driver attack surface from the outside world.

B. Execution Indexing

Our trace analysis for finding interrupt handlers relies on *execution indexing* [48], a technique for aligning pairs of program traces that marks points where they diverge and (importantly) reconverge. Because program traces may diverge for trivial reasons (such as an extra iteration of a loop), it is important in practice that we have a technique for re-identifying alignment after a divergence in order to focus only on large divergences. We briefly describe the core algorithm here, and then discuss our modifications to execution indexing to support whole-system traces on embedded firmware; we refer readers to Xin et al. [48] for a full treatment of execution indexing.

An execution index (EI) uniquely identifies a point in an execution and allows it to be compared across different execution traces. It uses a stack data structure that identifies the execution context of a basic block of code, e.g., how many conditional branches were encountered before the basic block was reached. Whenever a new code context is encountered, such as a function call or conditional branch, EI pushes onto the stack its address and the location of its immediate post-dominator. Because the immediate post-dominator of a basic block is, by definition, the earliest node through which every path from that block to the exit must pass, it identifies the end of an execution context (Johnson et al. [29] provide the helpful analogy that the closing curly bracket in C is an example of an immediate post-dominator). When the immediate post-dominator at the top of the stack is encountered in the trace, the stack entry is popped.

To compare two traces, EI starts by assuming they are initially aligned, with two empty EI stacks for both traces. It then steps through the two traces in tandem, updating the EI stacks at every step. Whenever the two EI stacks disagree,

[1] In practice, interrupt handlers will often queue the I/O and handle it later in a *bottom half*, in the interest of keeping interrupt latency low. However, in this paper we are only concerned with identifying the initial per-peripheral interrupt handlers, so this detail is not essential.

algorithm marks it as a control flow divergence, logs the divergence point, and enters the disalignment state. After that, assuming the deviated trace has entered a nested context which causes its EI stack to increase, the algorithm will try to realign the traces by stepping through the trace with the larger EI stack until the EI stacks agree once more, at which point the traces are considered re-aligned.

C. Assumptions and Usage Scenarios

We assume that (1) the analyst has access to a physical device for analysis; (2) that the analyst can upload new, modified firmware to the device; and (3) that the analyst can capture a snapshot of the device's CPU and memory at runtime.

Assumptions (1) and (2) are needed in order to actually make the modifications to the firmware and to validate the results. The third requirement is needed because IRQ handlers are typically dynamically registered during driver initialization. As a result, static analysis of a firmware image to uncover the handlers is unlikely to succeed, and so our system uses dynamic analysis to explore interrupt handling code. Dynamic analysis in embedded systems is generally considered a difficult and unsolved problem that requires either *rehosting* [22] or hardware-in-the-loop emulation [49]. Our system uses the CPU and memory state of the embedded device to resume execution inside a software emulator and systematically explore paths through the interrupt handling code by fuzzing the memory-mapped registers of the emulated interrupt controller.

Although this requirement may initially seem onerous, in fact it does not require any additional capabilities aside from those needed for software debloating: if one has the ability to rewrite the firmware of an embedded device, one can insert instrumentation that captures the CPU and memory snapshot as well. In our prototype, we collect memory and CPU snapshots using JTAG, QEMU, and a custom kernel module, depending on the target device.

Signed Firmware. Some devices now implement measures to prevent any modification of firmware by anyone aside from the vendor, using, e.g., signed firmware and secure boot [5]. This poses an obstacle to systems such as IRQDebloat, which aim to allow users to modify the firmware of their devices to improve security. However, we note that this limitation affects any system based on firmware modification. And in many cases these limitations need not be fatal: there is an active hobbyist community around modifying embedded and IoT devices to better suit their owners' needs by "jailbreaking" their devices and loading modified firmware [2].

Target Audience. We envision IRQDebloat as being useful to technically sophisticated users who need to deploy embedded or IoT devices and wish to modify the firmware without cooperation from the original firmware provider (in particular, we do *not* assume that the user has access to the source code for the firmware or a data sheet for the SoC). Examples of such users include businesses and government agencies, who typically cannot request custom changes from a vendor but may have security requirements that preclude the use of some functionality. Other possible users include hobbyists wishing to secure their own devices, technical users who wish to continue running devices past their end of life dates, and even system integrators, who may have the ability to create authorized firmware images but lack access to source code or detailed specifications for individual components.

III. Design

Figure 2 shows the high-level design of our system. In the initial stage, we capture a snapshot of the embedded device's CPU and memory state, and migrate it into a full-system emulator (PANDA [21]). Then, in PANDA, we trigger an interrupt and then fuzz the memory-mapped I/O (MMIO) registers to collect traces. Next, to discover individual handlers, we analyze the collected traces with differential slicing [29]. A given handler can be disabled by patching the firmware to replace the handler with a no-op function, which effectively causes the firmware to ignore input from that peripheral. Once the candidate handlers have been identified, the one corresponding to undesired functionality can be identified by a process of elimination: we can disable them one by one and boot the device to see if the undesired functionality has been disabled. When the peripheral's handler is identified and disabled, the attack surface from the peripheral's driver code is effectively closed off from the outside world.

A. Challenges

IRQ handler identification in firmware is not easy. Challenges arise from both the general difficulty of binary analysis as well as particular code patterns found in operating systems' implementations of IRQ handling.

1) Hardware Diversity: The first challenge is the diversity of interrupt controller hardware in embedded systems. Although there are some standards (e.g., the ARM Generic Interrupt Controller (GIC) specification [7] or the Nested Vectored Interrupt Controller (NVIC) [6]), vendors are free to ignore them (and many do). In our evaluation, we found that of the seven tested devices only one exclusively used the standard ARM GIC. As each interrupt controller can define its own protocol for tasks like retrieving the current interrupt number or acknowledging an interrupt, our analysis must be agnostic to particular interrupt controller hardware.

The need to be agnostic to different interrupt controllers poses some challenges for our analysis. For example, during the course of normal interrupt handling the OS will generally *acknowledge* the interrupt, which tells the interrupt controller to stop signaling a pending IRQ to the CPU. However, since the protocol for interrupt acknowledgment is not standardized, our emulator does not know when to stop signaling a pending IRQ, which can be problematic for some operating systems, including VxWorks. We develop a set of heuristics for determining when we should mark an interrupt as finished that allow IRQDebloat to explore the handlers through fuzzing.

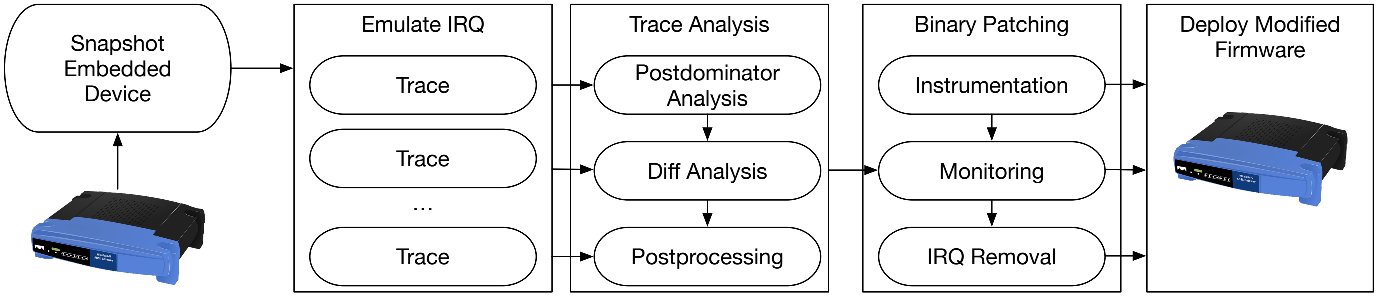


Fig. 2. Top-level workflow

2) Binary Analysis: Accurately identifying interrupt handlers without symbol information is a challenging binary analysis problem that currently relies on manual reverse engineering. From the perspective of the operating system, interrupt handlers are normally wrapped inside device-specific data structures with dynamically registered function pointers, and the handler dispatching routines make heavy use of loops to fetch device-specific register values and to match against the registered device drivers. For devices that require extra controllers, such as GPIO or I2C devices, similar dispatching subroutines may be chained under a top-level dispatching entry that binds the handler to the corresponding IRQ register values. In addition, some embedded operating systems such as RiscOS make heavy use of handwritten assembly code, which can break binary analysis heuristics such as function identification. This pervasive use of function pointers, nested loops, and chained handlers poses a significant challenge for automated static binary analysis.

Listing 1 shows an example of a GPIO interrupt dispatch routine in the Linux drivers for the Romulus BMC platform. First, the top-level IRQ status register (AVIC_IRQ_STATUS) must return 0x00100000 to enter the chained GPIO bus interrupt handler. Then `aspeed_gpio_irq_handler` is invoked to dispatch to the registered device interrupt handlers (keyboard and sysfs) based on the GPIO controller register values. The number of loops (banks) could go as high as 8, while every internal loop (`for_each_set_bit`) checks all 32 bits with the actual registered devices.

To help address the challenges of binary analysis, our trace analysis augments the static information from the binary with dynamic information collected from traces.

Listing 1 Linux GPIO driver code for Romulus BMC.

```
static void aspeed_gpio_irq_handler(
  struct irq_desc *desc
) {
  ...
  banks = DIV_ROUND_UP(gpio->chip.ngpio, 32);
  for (i = 0; i < banks; i++) {
    const struct aspeed_gpio_bank *bank = &aspeed_gpio_banks[i];
    reg = ioread32(bank_reg(data, bank, reg_irq_status));
    for_each_set_bit(p, &reg, 32) {
      girq = irq_find_mapping(gc->irq.domain, i * 32 + p);
      generic_handle_irq(girq);
    }
  }
  ...
}
```

3) Fuzzing Challenges: To make use of dynamic information during trace analysis, we collect traces by fuzzing the memory-mapped I/O values returned by peripherals on the embedded system in order to explore the IRQ handling code of the operating system. However, we cannot simply use an off-the-shelf fuzzer for this, as the code patterns involved in embedded IRQ handling are problematic for standard fuzzing techniques.

An example of a problematic code pattern taken from the Linux kernel code for the Samsung Exynos 4210-based NURI platform is given in Listing 2. During interrupt handling, the OS updates the system timer. However, the code to read from the timer repeatedly reads from the same 32-bit memory-mapped register (`EXYNOS4_MCT_G_CNT_U`) and cannot proceed until it reads the same value twice in a row. For a fuzzer that generates random values, the probability of generating the same 32-bit value twice in a row is low ($\approx 2^{-16}$), so the timer update function impedes fuzzer progress.

To address this challenge we introduce a set of fuzzing techniques that are tailored specifically to the patterns we find in IRQ handling code. This includes systematic exploration of common representations for IRQ numbers, and a "consistent I/O" mode that remembers the most recent value generated for a particular MMIO address and probabilistically returns the same value for future reads.

Listing 2 Nuri Linux timer register access

```
static cycle_t exynos4_frc_read(struct clocksource *cs)
{
    unsigned int lo, hi;
    u32 hi2 = __raw_readl(reg_base + EXYNOS4_MCT_G_CNT_U);
    do {
        hi = hi2;
        lo = __raw_readl(reg_base + EXYNOS4_MCT_G_CNT_L);
        hi2 = __raw_readl(reg_base + EXYNOS4_MCT_G_CNT_U);
    } while (hi != hi2);
    return ((cycle_t)hi << 32) | lo;
}
```

B. Snapshot Collection

Our prototype can use JTAG, an emulator like QEMU (if there is support for the target platform), or code running on the device (i.e., a kernel module) to collect snapshots. We collect physical memory and registers from the device, and load them into PANDA to continue emulation. Running the firmware in PANDA gives us the ability to control low level hardware behaviors (i.e., triggering interrupts) and monitor the execution environment at the same time. We are able to simulate interrupts and provide responses to MMIO reads, while inspecting processor states and collecting full execution traces.

C. Trace Collection

Once the snapshot is loaded into PANDA, the trace collector triggers an interrupt in the emulator, which causes the emulated CPU to switch into IRQ mode and jump to the architecture-defined entry point for interrupt handling. From there, we log the address of every basic block executed by PANDA. Whenever the firmware reads from a memory address outside of RAM, we supply a fuzzed MMIO value. After tracing is complete, we reset the emulator state and try another sequence of fuzzed MMIO values.

One challenge we face is deciding when to terminate tracing for a particular execution. Intuitively, we want the trace to be long enough to capture the execution of the handler for a particular IRQ, but short enough to allow performant fuzzing. Tracing until the CPU leaves IRQ mode is one approach that seems intuitively appealing, but in practice we find that some operating systems (including FreeBSD, VxWorks and Linux) leave IRQ mode well before they actually execute any handlers. Moreover, because many of the values we provide via MMIO are outside the range that actual hardware could produce, some traces may get stuck in infinite loops or trigger crashes or other errors in the emulated firmware.

Leaving IRQ mode is also problematic because, as mentioned above, we do not know when we should acknowledge the interrupt (i.e., clear the IRQ pending flag in the emulator). If an operating system re-enables interrupts before executing the actual IRQ handler, this will cause execution to repeatedly return to the top-level handler without making any progress. As a workaround, we acknowledge the interrupt after 10 basic blocks of code are executed in the emulator.

We impose a maximum trace length of 100,000 basic blocks during tracing so that we can make forward progress in exploration. Empirically, we have found that this threshold is sufficient to discover valid handlers, and allows the fuzzing stage to complete relatively quickly.

D. IRQ Fuzzing

During fuzzing, IRQDebloat attempts to enumerate the IRQ handlers on the system by triggering an interrupt and then responding to MMIO reads with values that are likely to be interpreted as different IRQ numbers by the firmware. This can be considered a type of fuzzing; however, the goal is not to uncover bugs but rather to simply explore the space of possible handlers. We assume that any memory read that falls outside of the RAM region should be treated as MMIO, and provide fuzzed values. MMIO writes are silently ignored.

Our IRQ fuzzer component uses a coverage-guided, generation-based approach. Within each generation we start with a set of $seeds$ (initially empty) where each seed $s = \{v_1, v_2, ..., v_k\}$ is a sequence of MMIO values. We form new candidate seeds by adding new MMIO values v_{k+1} to s from the patterns described in Section III-D1 to get $s' = s \| \{v_{k+1}\}$. We then use the trace collector to check if this new seed s' uncovers any new code; if so, we add it to $seeds$ and save the trace for further analysis. After each trace we reset the emulator to the snapshot state. A pseudocode version of this algorithm can be found in Appendix C.

If the provided sequence of MMIO values is exhausted during tracing before the trace length limit is reached, we return random MMIO values. During this stage, we can optionally enable a *consistent I/O* mode. In this mode, the values returned for a particular MMIO address will, with a configurable probability (currently 80%), have the same value. This is needed because some drivers (such as the timer in Listing 2) repeatedly poll a memory-mapped I/O address and check whether the value seen is the same, presumably as a workaround for spurious values returned by the hardware.

1) Fuzzing Patterns: Because there is a great deal of diversity in embedded interrupt controllers, we adopt a hybrid fuzzing strategy that tries common patterns used by known interrupt controllers as well as random values. We developed these patterns based on manual exploration of three initial targets (the Raspberry Pi, BeagleBone, and Romulus) and found that they worked well on the remaining targets in our evaluation set; we therefore expect that these patterns will generalize well to other embedded devices.

We implemented four patterns for MMIO inputs:

- **ints** provides integers $i : i \in 1..255$; this enumerates possible IRQs for controllers that store pending interrupts as an integer.
- **bitwin** provides a sliding window of k one bits in each possible bit position, for $k : k \in \{1, 2, 3, 4\}$; this enumerates possible IRQ numbers on controllers that indicate pending IRQs in a 32-bit bitset.
- **random** provides randomly chosen 32-bit values.
- **pattern** provides constants like 0, -1, 0xf0f0, etc.

These patterns are designed to give the fuzzer the benefit of domain knowledge (with the **ints** and **bitwin** patterns) while providing enough randomness to generalize well outside of what we have seen before (**random** and **pattern**). We show in Section V-B that these strategies are sufficient to thoroughly explore seven different models of interrupt controller across multiple operating systems.

E. Trace Preprocessing

Our coverage-guided fuzzing allows us to fully explore different branches in the IRQ dispatching, but the traces obtained are somewhat noisy. During interrupt handling, operating systems may take the opportunity to do other bookkeeping such as updating timers, incrementing performance counters, etc. Moreover, the fuzzer will produce many I/O values corresponding to invalid IRQs, so many of our traces will include error handling code and debug messages (which may include interactions with other peripherals like the serial UART).

In the trace preprocessing stage, we attempt to eliminate common sources of noise in two ways. First, we attempt to identify the subset of I/O sequences from fuzzing that relate to the interrupt controller. Starting from the collected IO sequences from the fuzzing stage, we regroup the IO sequences into small decoupled sequences, deterministically replay the IO sequences, and use these replayed traces for the differential analysis. To decouple the IO sequences, we keep a stack structure for MMIO addresses by pushing new IO addresses, and popping the stack until an already seen MMIO address is out. We log the stack at the same time, as it should represent the IO sequences in one loop/IRQ dispatch. The assumption is that the IO addresses related to IRQ handling are only read once during the time of processing one IRQ exception. In the rare case that an MMIO address is read multiple times during one interrupt, we merge the IO sequences with same MMIO address when they also have the same recorded IO value.

Second, we deduplicate traces by computing a hash of the sequence of basic block addresses in each trace. Since our trace alignment operates on pairs of traces, the amount of time needed grows as the square of the number of traces; thus, it is important to remove redundant traces. In our experiments, deduplication typically reduces the size of the trace dataset from thousands of traces to a few hundred.

F. Trace Analysis

We adapt the trace alignment algorithm from differential slicing [29] (which in turn is based on execution indexing [48]) to analyze the collected traces in order to infer the different IRQ handlers. The intuition is that since all the IRQ interrupts have the same entry point, they should share a common trace prefix before diverging at certain point into device specific code. By comparing pairs of traces from the fuzzing process, we can eventually find all the IRQ-specific divergences.

We found that the standard differential slicing algorithm required several modifications to make it suitable for analyzing whole-system traces from IRQ processing on embedded devices. We detail these changes in Section IV-D.

G. Instrumentation

Our instrumentation engine aims to be agnostic to the binary target file format. To achieve this, we use the memory and CPU dump (in order to identify where the code will be loaded at runtime), the list of handlers to disable, and the target binary to instrument. To actually disable an IRQ, we overwrite the handler with a dummy function that bypasses the actual handling code and returns immediately. At the moment, this still requires a small amount of manual reverse engineering to identify the correct return value for IRQ handlers on each OS.

To patch the handlers, we first translate the virtual address of the handler into a physical address. Next, for every patch point, we extract a small amount of data around the physical address from the memory dump to form a signature. Using this signature, we search through the kernel binary for a matching offset so we can statically instrument it. Although this approach is unlikely to generalize to cases of packed firmware, we consider the general problem of firmware unpacking and modification to be outside the scope of this paper.

IV. Implementation

In this section we describe our implementation of IRQDebloat in more detail. The fuzzing component of IRQDebloat consists of around 700 lines of C/C++ code in the form of plugins for the PANDA dynamic analysis platform [21]. The trace processing and binary analysis is implemented as a Binary Ninja plugin in 1100 lines of Python, and the firmware patching is implemented with 300 lines of Python.[2]

A. Snapshot Collection

Our snapshot implementation currently supports JTAG, live system dump, and QEMU-based snapshot acquisition for 32-bit ARM and MIPS platforms. It acquires snapshots using either a patched OpenOCD 0.10.0 (in the case of JTAG), a patched LiME [3] kernel module, or `gdb` in the case of QEMU.

Correctly restoring a snapshot in the emulator also requires that we collect all CPU registers needed for full system execution. This can be challenging on architectures like ARM, which uses banked registers (i.e., some registers have different values that are saved and restored depending on the CPU mode). We modified OpenOCD to capture banked registers by setting the CPU's mode (using the status bits in the CPSR register) to each of the available modes, and then dumping the registers from that mode. We also need to collect many coprocessor registers. Due to space constraints, the full list of registers we collect for ARM and MIPS can be found in Appendix A.

To dump physical memory, we modified OpenOCD's `dump_image` command to use physical rather than virtual addresses. For QEMU we use the `pmemsave` command in the QEMU monitor. And to dump memory on a live system, we rely on the LiME kernel module [3]. Dumping physical memory requires that we know the start and size of RAM for the embedded device. However, this information can usually be obtained via JTAG or QEMU's `info mtree` command.

B. Trace Collection

To collect traces, we created an empty machine model for PANDA that has an ARM or MIPS CPU, no peripherals, and a block of RAM at a configurable address and size. We implemented a PANDA plugin that loads a saved snapshot and triggers an interrupt (using QEMU's `cpu_interrupt` function). From there, the fuzzing plugin forks child processes, generates the fuzzed values in response to MMIO reads by registering a callback for `PANDA_CB_UNASSIGNED_IO_READ`, and reports information about basic block coverage. For embedded devices that have MMIO regions that overlap with RAM, we could modify the machine model to create an I/O memory region at the appropriate location; however, we have not encountered this case in our testing.

[2] Source line counts generated using David A. Wheeler's 'SLOCCount'.

C. Trace Analysis

1) Preprocessing: As described in Section III-E, we preprocess traces by deduplicating them using the hash of the basic block sequence, and minimizing them to create shorter traces that contain, to the extent possible, only the IRQ handling code.

We also automatically infer a list of potential I/O addresses that correspond to polling loops, such as those used by timers and UARTs. These inferred I/O addresses will be ignored when we regroup I/O sequences, and will be replayed with specific values (randomly chosen from 0 or -1) during the I/O replay. These values were chosen by examining the code of several timer and UART drivers and identifying values that minimize the amount of polling needed among the hardware seen in our evaluation corpus. An example of this sort of problematic code is given in Listing 3; here, the FreeBSD kernel repeatedly polls the UART's FIFO status register until the FR_TXFF bit is clear. Returning 0 allows this loop to be bypassed, resulting in a shorter trace.

Listing 3 PL011 UART driver in the FreeBSD kernel.

```
static void
uart_pl011_putc(struct uart_bas *bas, int c)
{
    /* Wait when TX FIFO full. Push character otherwise. */
    while (__uart_getreg(bas, UART_FR) & FR_TXFF)
        ;
        __uart_setreg(bas, UART_DR, c & 0xff);
}
```

To identify timer and UART peripherals, we identify any MMIO address that has a significantly higher number of appearances than the first MMIO read address (which corresponds to the top-level IRQ source register). Empirically, we found that a threshold of around 10 times higher than the appearance number of the first MMIO read address works well to identify this kind of polling code.

2) Postdominator Computation: The preprocessing stage augments the trace with immediate postdominator information so that each basic block address in the trace becomes a pair $(addr, ipdom)$.

We first collect all functions and possible return nodes using static analysis to disassemble each function and create a CFG. In our implementation, we use Binary Ninja (2.4.2846) as our base disassembler. To help overcome the limitations of imprecise binary analysis, we guide the disassembly with the addresses from the traces. To compute postdominators we use the standard fast dominance algorithm from Cooper et al. [15].

In some cases, we found that the disassembler could miss basic blocks in a function. To overcome this, we split the CFG of such functions into multiple sub-CFGs and compute postdominators for the smaller CFGs. An example of this occurs when Binary Ninja fails to resolve a switch statement. Instead of trying to resolve the switch table, we mark the basic block as a return node, and the switch table targets (observed from the trace as the next instruction) as the start of a new function. The EI stack and immediate postdominators will be computed locally inside those functions without changing any divergence results.

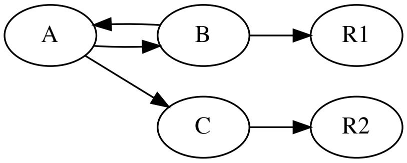


Fig. 3. Example CFG and two traces through the CFG: "A→C→R2" and "A→B→A→C→R2".

During preprocessing we also resolve inconsistencies between QEMU's translation blocks and actual basic blocks in the static CFG. For QEMU, a translation block is any straight-line code that terminates in a branch or call instruction. However, basic blocks in the static CFG may be split up further, since incoming edges to the middle of a block are not allowed.

To fix the issue, we check for missing basic blocks during the trace re-processing. For every address in the trace, we cross check the next address after its basic block ends, the outgoing edges of the basic block, and the next address in the trace, and determine whether we need to split the translation block into a true basic block. If so, we create a new trace entry corresponding to the start of the basic block.

D. Divergence Analysis

Our divergence analysis is an implementation of execution indexing [48], which we gave an overview of in Section II-B. To better handle the kind of traces encountered in embedded IRQ fuzzing, we made several modifications to the core algorithm, which we detail in this section.

1) Trace Realignment with Multiple Return nodes: One substantial step in trace analysis is to build an immediate postdominator mapping for every basic block inside the function. In the case of multiple return nodes in a function, a common approach is to create a fake return node sink, and add an edge from every actual return node to the fake return node, so that we can use the fake return node as a single root to compute the immediate postdominators.

However, during our experiments, we found this approach can sometimes produce unnecessarily large divergences. Consider the example CFG in Figure 3 and two traces through it, "A→C→R2" and "A→B→A→C→R2". If we create a fake node for R1 and R2, both A and B will have the fake node as the immediate postdominator. With the update of the EI stack, two traces will diverge at the first address, but since their immediate postdominators are both the fake return node, the two traces would only be able to realign at the fake return node after R2. As a result, we will miss the node C, especially if C dispatches to another IRQ handler. However, if we remove R1, and build the immediate postdominator based on R2, node A will be the immediate postdominator of B, and C will be the immediate postdominator of A, so we will still be able to realign at node C.

As a solution, we build separate postdominator mappings for each return node. During trace processing, we pick the appropriate postdominator mapping by looking ahead in the trace to see which return was actually invoked.

2) Trace Truncation: Whereas traces of traditional userland programs are typically complete (i.e., they all eventually reconverge at the program's exit point), our IRQ fuzzing traces are often cut short either because we reach the trace length threshold or because the execution hits some error during tracing. This means that some functions in the trace will have no return node, which is needed to identify the post-dominator at a function call. To address this problem, we detect trace truncation and promote the last basic block for that function in the trace to a return node.

3) Walking Misaligned Traces: When traces diverge, the original trace alignment algorithm favors shorter EI stacks, and only proceeds on the trace with a larger EI stack until they re-align. However, because our traces may be truncated, we have found that sometimes a shorter EI stack doesn't mean the trace is closer to the end.

Instead, our trace alignment first tries to align the traces by proceeding on the trace with the larger EI stack. If realignment fails, we re-try with the shorter EI stack. In the case where a divergence is encountered and both EI stacks are the same height, we try to proceed on both traces and then pick the alignment that re-converges earlier (i.e., has a smaller amount of divergence).

E. Postprocessing

The output from differential analysis is a list of divergence points, which may also contain unrelated divergence points in addition to the expected IRQ dispatchers. However, we can take advantage of the fact that IRQ handlers are registered dynamically to more precisely identify the handlers. Because handlers are registered dynamically, they must be stored by the OS as function pointers; thus, we can filter the results by including divergences where 1) the divergence is caused by an indirect call, and 2) the branch target is a function.

On ARM, indirect calls are generally made via the `blx` and `bx` instructions; MIPS uses `jr` and `jalr`. To check function targets, we reuse the results from trace preprocessing (which gives us an over-approximation of potential functions), and match the branch target address against the function start addresses. We will show in Section V-E that although this filtering strategy is simple, it suffices to eliminate almost all the false positives in our dataset.

F. Instrumentation

Our instrumentation modifies the target kernel to remove the handler corresponding to undesired hardware functionality. To do so, we must locate the function in the binary firmware image, and then overwrite it. To find the appropriate locations to patch in the static firmware image, we create a signature out of the bytes surrounding the hook site. Then, we scan the firmware image to locate the matching code. Our signature is 32 bytes, which we have found in testing to be sufficient to uniquely identify the code locations we need to patch.

TABLE I
PLATFORMS AND OPERATING SYSTEMS TESTED

Name	OS	SoC	Snapshot	IntC
RasPi	Linux	BCM2837	JTAG	BCM
RasPi	FreeBSD	BCM2837	JTAG	BCM
RasPi	RiscOS	BCM2837	JTAG	BCM
Beaglebone	Linux	TI AM335x	JTAG	AM335x
SABRE Lite	VxWorks	BD-SL-i.MX6	QEMU	GICv3
SABRE Lite	Linux	BD-SL-i.MX6	QEMU	GICv3
Samsung NURI	Linux	Exynos 4210	QEMU	GIC+Combiner
Romulus	Linux	AST2500	QEMU	ASPEED VIC
WRT54GL	Linux	BCM5352	JTAG	MIPS
SteamLink	Linux	MV88DE3108	KMod	GIC+APB

Our current prototype supports disabling handlers on Linux, FreeBSD, and RiscOS. In Linux, we patch the IRQ handler by overwriting the start with `mov r0, 2; mov pc, lr`, which sets the return value to `IRQ_WAKE_THREAD` (2) and returns. Similarly in FreeBSD, we set the return value to be `FILTER_SCHEDULE_THREAD` (4) and return. In RiscOS, we instead patch the IRQ handler with `bic r11, r11, 1; mov pc, lr`, which clears the `IRQDesp_Link_Unshared` bit and then returns. We determined these values by reading the kernel source for each OS, but in future work we hope to determine appropriate return values automatically.

V. EVALUATION

In this section, we evaluate IRQDebloat along two dimensions of generality: hardware diversity and firmware diversity. To demonstrate that our approach to interrupt handler identification works across diverse interrupt controller models, we test it against six ARM-based platforms and one MIPS platform across four different operating systems, detailed in Table I, for a total of ten different embedded configurations. By examining the Interrupt Controller (IntC) column, we can see that only one of our tested systems, the i.MX6-based SABRE Lite board, exclusively uses the ARM GIC standard, indicating that there is considerable diversity in embedded interrupt controller hardware.

To show that we can generalize across different operating systems, which may have different code patterns for IRQ dispatch, we evaluate on Linux, FreeBSD, VxWorks, and RiscOS. RiscOS in particular serves as a robustness test for our trace alignment and IRQ handler identification, as the operating system is quite old (dating to 1987) and written almost entirely in ARM assembly.

In each case, we begin by establishing ground truth. For hardware, this means examining the datasheets and open-source firmware to understand how the interrupt controller works at the hardware level. For the operating system evaluation, we locate the handler registration code in each OS and then add logging code to collect a trace of all interrupt handlers known to the OS.

Next, we evaluate the fuzzing, preprocessing, and divergence analysis components of IRQDebloat. We compare the identified handlers to the ground truth on each platform to identify false negatives (i.e., registered IRQ handlers that are

TABLE II
REGISTERED IRQ HANDLERS FOR EACH OS. † INDICATES A THREADED HANDLER.

	RasPi Linux	**RasPi FreeBSD**	**RasPi RiscOS**	**BeagleBone Linux**	**Romulus Linux**	**Nuri Linux**	**Sabre Linux**	**Sabre VxWorks**	**WRT54GL Linux**	**SteamLink Linux**
IPI	1	5	0	0	0	0	0	1	0	0
Mbox	1	1†	1	2	0	0	0	1	0	0
SYS_Timer	0	0	1	1	1	2	1	2	0	1
PMU	1	1	0	1	0	0	1	0	0	0
USB	3	1	1	1+1†	1	0	0	0	0	1
GPIO	0	2	0	1	2	1	1	1	0	0
MMC/SD	1	1†	1	1+1†	0	0	1+1†	0	0	1
Loc_Timer	1	1	1	0	0	0	1	0	1	0
UART	1	1	1	1	1	1	1	1	1	1
FrameBuf	1	1†	1	0	1	0	3	0	0	1
Video	1	1†	1	1+1†	1†	0	6+1†	1	0	4
DMA	1	0	7	3	0	0	1	1	0	1
Eth	0	0	0	2	1	0	2	1	2	1
I2C	0	0	0	1+1†	1	0	1	0	0	1
SPI	0	0	0	1	0	0	1	0	0	0
Chained/Virtual	3	1	0	2	6	6	4	0	0	4
Other	0	0	0	3+3†	1	0	3+4†	1	0	1

not found during fuzzing or analysis) and false positives (i.e., non-handler code mistakenly identified by the divergence analysis). We also demonstrate that our preprocessing successfully reduces the size and number of traces we need to analyze.

Finally, we perform two case studies with platforms in our evaluation dataset. To understand the impact of disabling IRQ handlers on a real system (the Raspberry Pi), we systematically evaluate the effect of disabling each handler found across three operating systems (Linux, FreeBSD, and RiscOS). Next, we give a practical demonstration of IRQDebloat's ability to reduce attack surface with the popular Steam Link streaming device, showing that disabling the IRQ handler for the WiFi/Bluetooth functionality succesfully blocks a Bluetooth exploit. We then systematically measure the attack surface reduction (ASR) on this device by counting how many CVEs would have been blocked over the past five years by disabling each interrupt on the device.

A. Hardware Ground Truth

To characterize the types of interrupt controllers in our evaluation corpus, we analyzed the datasheets, kernel source code, and (for devices supported by QEMU) the QEMU emulated peripheral source. Broadly, we find two main ways of identifying a pending interrupt i: either by setting the i-th bit in a 32-bit register, or by returning the integer i. This validates our fuzzing strategy, which systematically enumerates small integer values as well as setting each possible bit in each 32-bit MMIO register.

The Raspberry Pi has the most complicated IRQ dispatch hardware among the tested platforms. The BCM2837 SoC has 4 CPU cores, and each core has an independent 32-bit IRQ source register to signal which peripheral device raised the IRQ. Every bit of the IRQ source register is used to determine one specific device, and device IRQs can be configured to route to any one of the cores' IRQ source register, including Inter-Processor Interrupts (IPI), Performance Monitor Unit (PMU), Local Timer etc. Among these, there is also a secondary interrupt controller from the GPU (interrupt $1 \ll 8$, or 256) chained to the global interrupt controller, which has three further 32-bit IRQ pending registers that signal which global device raised the IRQ, such as global timer, UART controller, GPIO controller etc. The masked bits of these 3 registers in combination determine the source device during an interrupt.

The ASPEED Romulus has a similar, albeit simpler, hardware design. It has two 32-bit source registers at the top-level, and each bit of the registers signals one particular device. Similarly, users can further register chained IRQ handlers under each device to dispatch device specific interrupts. In our setup, we have a I2C controller that further dispatches one hardware IRQ in a bit-masked 32-bit register, and a GPIO controller that dispatches two registered IRQ handlers through 8 MMIO registers, each of which can indicate up to 32 interrupt sources (although in practice only the first 29 are enabled on the Romulus).

Three devices in our evaluation dataset (SABRE Lite, Samsung NURI, and Steam Link) use the standard ARM GIC [7], which is thoroughly documented by ARM. To obtain the IRQ number, the GIC provides a single 32-bit MMIO register, `GICC_HPPIR` which returns the highest-priority pending interrupt in its lowest 10 bits (for a total of 1024 possible IRQ numbers, although IRQs 1020-1023 reserved for signaling conditions such as spurious interrupts). In addition to the GIC, the NURI and Steam Link have additional secondary interrupt controllers. The NURI has a "combiner" peripheral that allows multiple peripherals' interrupts to be grouped into a single GIC input and supports five different groups; each group has a corresponding 32-bit MMIO register that indicates which member of the group raised the interrupt using a bit set. The Steam Link has two secondary Synopsys DesignWare interrupt controllers on the Advanced Peripheral Bus (APB). Each APB interrupt controller has a 32-bit register where each bit represents a separate interrupt.

Unlike many of the ARM systems we examined, the MIPS-based Linksys WRT54GL has a well-documented way of handling interrupts. The interrupts are signaled in the CP0 Cause register through the IP0–IP7 bit fields (for a maximum of 8 interrupts). Each interrupt can be independently enabled/disabled in the CP0 Status register through the IM0–IM7 mask bits. Notably, MMIO is not used, meaning that we do not need to actually fuzz the MMIO values to collect traces. More complex MIPS devices may include dedicated interrupt controller peripherals (e.g., the MIPS Malta board uses a PC-style Intel 82C59 interrupt controller, and the MIPS Boston board uses the MIPS Global Interrupt Controller (GIC)).

Finally, the BeagleBone has the simplest design compared to the other boards. It reads a 7-bit integer value (128 total possible IRQs) from the interrupt controller register, and dispatches directly using the hardware IRQ number. We did not encounter any chained IRQ handlers with the BeagleBone in our setup.

B. OS Ground Truth

To verify our analysis results, we first need to obtain the ground truth of how many IRQ handlers are actually registered, and how many of those registered handlers are actually used at the runtime.

For Linux and FreeBSD, which are open-source, we instrumented the kernel source and added print functions to the interrupt registration APIs. We instrument irq_request_irq, irq_request_percpu_irq and request_threaded_irq in the Linux kernel and intr_setup_irq in the FreeBSD kernel. For RiscOS, which has a fairly simple design for its IRQ handling, we studied the code for the interrupt handling routine, and then located and parsed the device and IRQ data structures from the memory dump to enumerate the registered handlers. Finally, for VxWorks, we reverse engineered the interrupt handling code (aided by the presence of debug symbols in our evaluation image) and then used gdb to enumerate the list of handlers from a running system.

Table II shows the listing of registered IRQ handlers we found in each operating system. Note that threaded IRQ handlers are not included in our evaluation. We discuss this limitation further in Section VI.

C. Fuzzing Evaluation

Different interrupt controllers and implementations of IRQ dispatching across operating systems may have different behaviors for the same sequence of fuzzed MMIO values. In this section, we evaluate how effective our fuzzing strategies (described in Section III) are at covering the IRQ handlers for each platform.

Figure 4 shows the IRQ handler coverage for each platform as fuzzing proceeds. In our test set, we see that IRQDebloat's fuzzer uncovers almost all handlers within 3 hours; in one case (Linux running on SABRE Lite) it takes up to 24 hours to uncover the final handler.

By comparing to the ground truth for each platform, we find only two handlers that the fuzzer is unable to uncover: mxc_epdc_irq_handler and vdoa_irq_handler in Linux for SABRE Lite. On further investigation, we found that these are masked (i.e., disabled) by the operating system and cannot be triggered in our configuration. In other words, the fuzzer successfully finds all of the reachable handlers in our test set.

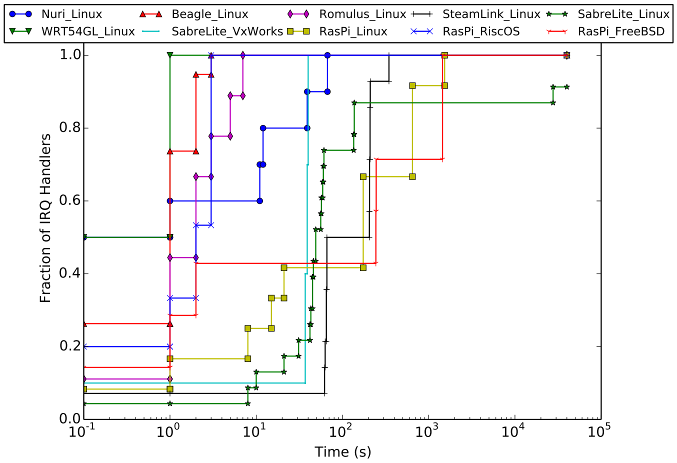


Fig. 4. Fuzzing coverage over time.

TABLE III
EFFECT OF TIMER/UART POLLING REMOVAL

	# Blacklisted MMIO	Before	After
Romulus Linux	4	490,446	1087
Sabre VxWorks	0	66	N/A
RasPi FreeBSD	1	25,583	647
RasPi RiscOS	0	50	N/A
Beagle Linux	2	2,166	184
RasPi Linux	1	1,277	681
Nuri Linux	4	41,540	266
Sabre Linux	3	21,293	86
SteamLink Linux	0	15,106	N/A

D. Preprocessing

The goal of our preprocessing stage is to simplify the traces by minimizing common sources of noise (e.g., timer/UART polling) and extracting shorter I/O sequences that allow handlers to be found earlier. To evaluate the effect of this preprocessing, we measured the minimum, maximum, and average position in the trace for each handler on each platform. Across the platforms tested, we find that I/O sequence grouping reduces the average number of blocks in the trace needed to reach a handler from 47,171 to 6,935.

We also investigate the effect of identifying and blacklisting polling loops. Table III shows the number of grouped I/O sequences before and after applying the IO blacklist. (We omit WRT54GL here since it does not use MMIO for interrupt dispatching.) Each grouped I/O sequence is replayed to create a trace for the divergence analysis, so reducing the number of such sequences is critical for performance. We find that our heuristic effectively reduces the number of sequences we need to consider by between 1.9X and 451X (151X on average), indicating that it provides significant savings.

E. Divergence Analysis Results

We report the results of our divergence analysis in Table IV. In addition to false positives and negatives, we also include the number of chained handlers (“Chain”) and the number of default handlers (“Default”). These represent handlers that

TABLE IV
DIVERGENCE ANALYSIS FALSE POSITIVES AND NEGATIVES

	Total	FN	FP	Chain	Default
Beagle Linux	23	0	0	2	2
Nuri Linux	11	0	0	6	1
RasPi FreeBSD	12	0	0	1	0
RasPi Linux	14	1	0	3	0
RasPi RiscOS	16	0	0	0	1
Romulus Linux	19	0	2	6	2
Sabre Linux	28	2	0	4	1
Sabre VxWorks	12	0	2	0	0
WRT54GL Linux	4	0	0	0	0
SteamLink Linux	21	0	3	4	1

do not appear in our ground truth (and so would normally be considered false positives) but which we feel can justifiably considered real handlers. **Chained** handlers are virtual handlers that dispatch to more specific handlers. We do not consider these false positives, since a user may wish to use such handlers to disable multiple peripherals at once. In some cases, there may only be a single sub-handler registered under a chained handler. In this case IRQDebloat will only find the top-level handler in the chain, since there will be no further divergences. We consider this a true positive, since disabling the top-level handler will still disable the desired peripheral. **Default** handlers similarly do not appear in our ground truth, as they are statically registered by the OS as a fallback when no other handler is suitable. These are found by our divergence analysis, since invalid IRQ numbers will diverge from valid IRQs. We do not include these as false positives.

Aside from chained and default handlers, we have two false negatives for Sabre LITE Linux, two false positives for Sabre LITE VxWorks, two false positives for Romulus Linux, and three false positives for Steam Link Linux. The two false negatives are missed during the fuzzing stage (as mentioned in Section V-C) and so do not appear in any trace.

The false positives in Sabre LITE VxWorks and Romulus Linux are caused by internal callback functions inside specific IRQ handlers. For Sabre LITE VxWorks, the false positive is located at `__udivmodsi4`, an LLVM compiler intrinsic for the integer divide and modulus operation, which has a switch statement at the end of the function for optimization. Romulus Linux has two callback functions registered under its UART interrupt handler (serial8250_interrupt), which could invoke either aspeed_vuart_handle_irq or serial8250_default_handle_irq. The Steam Link false positives turn out to be caused by Linux softirq tasklets (callbacks which can be scheduled by drivers and run at the end of IRQ handling in irq_exit).

While not ideal, false positives are not particularly harmful in our setting. A false positive will mean that the user has one more handler to check (which just means booting the device and seeing if the unwanted peripheral has been disabled).

F. IRQ Monitoring and Removal

To evaluate the IRQ monitoring and removal on a real device, we used the results of our analysis to instrument three operating systems (Linux, FreeBSD, and RiscOS) running on a Raspberry Pi. We selectively disabled all discovered IRQ handlers on each operating system as shown in Table V. For the USB and Video IRQ handlers in RiscOS (indicated with a † in the table), we found that the handlers could not be disabled during boot, so we used our instrumentation engine to add code that waits until the system has booted (using a simple counter) and then disables the handler at runtime. The "N/A" entries in the table indicate that the OS does not register a handler for that device. Finally, ☠ entries indicate that the device cannot function properly with that IRQ disabled.

TABLE V
IRQ HANDLER REMOVAL RESULTS.

	Linux	FreeBSD	RiscOS
UART	✓	✓	✓
USB	✓	✓	✓†
Video Controller	✓	N/A	✓†
DMA	✓	N/A	✓
GPIO	N/A	✓	N/A
PMU	✓	✓	N/A
Timer	☠	☠	☠
IPI	☠	☠	☠

With the UART disabled, we lose all input/output through the serial port, while all the other parts (HDMI, USB etc.) are unaffected. When USB is disabled, we can still interact through the serial port, but we lose all control from USB-connected devices (keyboard, mouse etc.), as well as the Ethernet connection, which is internally connected to the USB bus on the Raspberry Pi. We did not see any effect from disabling the video controller interrupt (DMA) or Performance Monitoring Unit (PMU). The remaining handlers (timer and inter-processor interrupt (IPI)) are essential for device functionality and cannot be disabled.

G. Case Study: Bluetooth on the Steam Link

We use the Valve Steam Link to illustrate a real-world case of how IRQDebloat could be used to reduce attack surface. The Steam Link is based on the Marvell Armada 1500-mini (MV88DE3108) SoC and is used to stream games from a desktop PC to an external display via HDMI. It runs Linux and supports connectivity via WiFi, Bluetooth, and Ethernet. Our scenario considers a user who does not have any Bluetooth devices and wishes to disable this functionality to avoid Bluetooth exploits such as BleedingTooth [37]. Indeed, we found that the Steam Link in its default configuration is vulnerable to one of the BleedingTooth vulnerabilities, BadChoice (CVE-2020-12352; the kernel version used, 3.8.13, is too old to be vulnerable to BadKarma and BadVibes).

Although the Steam Link only accepts signed firmware and does not have JTAG exposed on the board, Valve has released information that allows users to log in as root on the device. This allowed us to create a snapshot using our own custom kernel module (for the CPU registers) and LiME [3] (for the memory dump). Root access also means that we can patch the kernel in memory in order to disable handlers, even though we are not able to modify the firmware image itself.

Our analysis finds a total of 21 handlers. We disabled them one by one and checked whether Bluetooth was still working on the device. If available, kernel symbols (e.g., `/proc/kallsyms` on Linux) can speed up this process, but are not required. After some analysis we determined that WiFi and Bluetooth functionality are both handled by the Marvell 88W8897 peripheral, which uses a shared interrupt on the SDIO bus for both protocols.

We opted to disable both WiFi and Bluetooth on the device by overwriting the shared SDIO handler. Although this is not ideal (we only *intended* to disable Bluetooth), we note that the Steam Link can still access the internet over Ethernet, so the device is still usable. We then checked whether it was still vulnerable to BadChoice by running the proof-of-concept exploit code, and confirmed that the exploit failed. Indeed, all Bluetooth connections to the device failed.

H. Quantifying Attack Surface Reduction

To more rigorously assess the potential attack surface reduction from disabling handlers on the Steam Link, we collected a list of all CVEs in the past five years (2016–2021) for which the Linux Kernel CVEs project [4] could identify a patch, excluding patches larger than 1MB (as these usually removed entire subsystems and would not allow precise matching). In total we found 978 CVEs that met our criteria.

Next we annotated each IRQ on the Steam Link according to whether it is exposed to external input and then manually identified which kernel source paths were potentially reachable from the interrupt handler. We included here both the driver code for the Steam Link's particular peripherals as well as code that handled protocols accessible only via those peripherals (e.g., the Bluetooth protocol stack). We then matched these source paths against our CVE patches to determine which CVEs could have been blocked by disabling the handler. We used the National Vulnerability Database's Attack Vector classification to exclude CVEs that could only be exploited locally by a process running on the device.

The results are shown in Table VI. We can see that there are indeed many vulnerabilities in the WiFi and Bluetooth drivers that would be prevented by disabling the handler (13 CVEs). Similarly, USB devices have been the source of many vulnerabilities, and disabling this interrupt would have blocked 31 vulnerabilities. This count is conservative in two ways. First, it does not consider vulnerabilities in the peripheral firmware itself. For example, the ThreadX-based firmware on the 88W8897 WiFi/Bluetooth peripheral is vulnerable to CVE-2019-6496; while disabling the IRQ for this peripheral would not prevent the *peripheral's* firmware from being exploited, it would prevent further escalation to the main application processor. Second, the attack surface reduction from disabling multiple IRQ handlers at once is not just additive: for example, disabling IRQs 49 and 56 (WiFi/BT and Ethernet) would disable all network access, preventing any vulnerabilities in the entire network stack from being exploited.

TABLE VI
STEAM LINK ATTACK SURFACE REDUCTION

IRQ#	Peripheral	Ext?	Source	#CVEs
32	Presentation Engine	✗	N/A	N/A
33	Presentation Engine	✗	N/A	N/A
38	GPU	✗	N/A	N/A
44	USB	✓	drivers/usb	31
49	Bluetooth (SDIO)	✓	drivers/bluetooth, net/bluetooth	7
49	WiFi (SDIO)	✓	drivers/net/ wireless/marvell/ mwifiex	6
50	NAND Flash	✗	N/A	N/A
51	Presentation Engine	✗	N/A	N/A
56	Ethernet	✓	drivers/net/ ethernet/marvell/ geth.c	0
57	Presentation Engine	✗	N/A	N/A
168	APB Timers	✗	N/A	N/A
176	Two Wire Serial I2C	✗	N/A	N/A
200	UART	✓	drivers/tty/serial/ 8250	0
208	Presentation Engine	✗	N/A	N/A
Total remotely exploitable kernel CVEs, 2016–2021				157

VI. LIMITATIONS AND FUTURE WORK

Non-IRQ Inputs. A core assumption we make is that unwanted input to the system is delivered via interrupts. It is possible that some systems just poll for input from a peripheral in a loop. In our experience thus far, this type of input handling is generally only used for very simple peripherals (e.g., simple sensors), but we hope to more thoroughly explore this type of input and find ways to disable it automatically in future work.

Shared and Threaded Handlers. There are currently some OS mechanisms for handling IRQs that we do not yet support. In particular, some peripherals are handled using *threaded* handlers, in which incoming IRQs are queued and then handled asynchronously in a different kernel thread. This can be particularly problematic when multiple devices share a single interrupt (the shared WiFi/Bluetooth interrupt on the Steam Link): although the individual peripherals do have their own handlers, they are only executed after the initial handler has ended, so IRQDebloat can only see and disable the initial top-level handler.

We believe that such handlers can be identified by continuing our fuzzing after the initial IRQ handler returns and then applying our divergence analysis to the code executed by the kernel scheduler. Since the same threaded handler will always be invoked for a given IRQ, the analysis should allow us to distinguish between threaded handlers and other kernel thread activity. However, we leave this to future work.

Manual Effort. Some parts of our system still require manual effort, which we describe and quantify here; note that the times given are for a user who is relatively skilled at reverse engineering and embedded device hacking. First, human effort is required to acquire a snapshot from the device and to figure out how to flash a modified firmware image. This is currently one of the more time-consuming pieces, as it depends on the device and whether any protection measures exist. For three of the four physical devices in our dataset, we were able to obtain a usable snapshot within 2-3 hours of work, but one device (the WRT54GL) took several days of work due to our unfamiliarity with MIPS.

It also requires human judgment to validate that the right peripheral has been disabled. The user must boot the device with the modified firmware and check whether the disabled peripheral responds to input. However, this does not require technical expertise (beyond what is required to modify the firmware) and only takes a few minutes per candidate handler.

Finally, identifying a return value for a disabled handler that satisfies the kernel's IRQ handling API currently requires manual reverse engineering for each operating system (around an hour per OS). One possibility for automating this step is to use the dynamic information from our traces to see what valid handlers return, and try those values.

Rehosting. Aside from addressing limitations of our system, we also believe that our techniques for automatically enumerating and identifying interrupt handlers could be useful for other embedded reverse engineering efforts, such as rehosting [22], [45]. When faced with a new embedded system, it can be challenging to even enumerate its peripherals and identify their memory-mapped register ranges. IRQDebloat could help automate this process, as each interrupt identifies a unique peripheral, and the interrupt handler usually contains code to interact with the device's memory-mapped registers. In addition, interrupt handling remains a challenge for automated rehosting systems; we hope to extend IRQDebloat to automatically create an emulated model of the target interrupt controller along with stub devices for each interrupt source that will need to be emulated.

VII. Related Work

As embedded and Internet of Things (IoT) devices have proliferated, academic research has turned its attention to the challenges of securing these systems. Embedded analysis is difficult due to the wide variety of hardware used, and the relative dearth of embedded security tooling.

One line of embedded research focuses on bringing dynamic program analysis to embedded systems by emulating them in an embedded environment. These include pure software emulation with inferred MMIO inputs or abstracted hardware interfaces [11], [14], [25], [35], [41], [18], [23], [34], [26] and hardware-in-the-loop hybrid emulation [49], [36], [44], [30], [32], [17]. Two recent surveys [22], [45] provide a more comprehensive picture of the state of the art in rehosting, and we refer interested readers there for more details.

Aside from dynamic analysis, some work has sought to analyze embedded firmware statically. FIE [20] lifts MSP430 source to LLVM and then symbolically executes it in KLEE [9], while Firmalice [43] symbolically executes binary firmware to identify potential backdoors. On the protection side, Cui and Stolfo proposed embedded *symbiotes* [19], in which embedded firmware is rewritten to inject a runtime security monitor. Our work also focuses on improving the security of deployed embedded devices, but does so by removing, rather than adding functionality.

Separately, interest has recently grown in how to eliminate unwanted features in existing software to reduce its exposed attack surface. This technique, known as *debloating*, has been applied to desktop and server software [12], [39], [38], [31], [28], mobile applications [47], [27], web applications [8], and even containers [40].

Work on debloating embedded firmware is somewhat more scarce. DECAF [13] and me_cleaner [16] attempt to remove unused functionality in UEFI firmware and the Intel ME BIOS, respectively. Specifically for Linux-based embedded systems, Lee et al. [33] removes unused syscall and driver code for hardware that is not present, and Chanet et al. [10] use link-time binary rewriting to remove unreachable system calls.

Closest to our own work is LIGHTBLUE [46], which debloats Bluetooth protocol stacks in embedded devices to reduce the attack surface exposed by Bluetooth. LIGHTBLUE allows individual portions of the Bluetooth protocol to be disabled by analyzing an accompanying profile. Although this allows for more fine-grained control over Bluetooth (IRQDebloat can only enable or disable the Bluetooth peripheral as a whole), it cannot be used to limit the attack surface caused by non-Bluetooth peripherals.

VIII. Conclusion

In this paper, we presented IRQDebloat, an automated technique for reducing the exposed attack surface of embedded devices by rewriting firmware to disable unwanted hardware functionality. To do so, we identified a key architectural feature—namely, interrupt handling—that acts as a bottleneck through which inputs enter the system, and then developed an automated dynamic analysis to precisely identify and disable interrupt handlers for unwanted peripherals. We believe that in addition to giving end users more control over the embedded hardware deployed on their networks, our techniques may also be useful for automated reverse engineering of embedded systems for security.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback, which helped make the paper much stronger. We also warmly acknowledge the contributions of NYU students Amandeep Singh, Kevin Kwan, Fernando Maymi, Casey McGinley, Birkan Erenler, and Hao Ke, who performed early exploratory work on IRQDebloat throughout course and independent study projects in 2016–2018. The work in this paper is supported by funding from the Office of Naval Research under ONR award N00014-15-1-2180 and the National Science Foundation under Grant No. CNS-1657199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research or the National Science Foundation.

Availability

The code and data used in this paper are available at:
https://github.com/messlabnyu/irqdebloat

References

[1] Cisco Meraki: Bluetooth low energy (BLE). https://documentation.meraki.com/MR/Bluetooth/Bluetooth_Low_Energy_(BLE).

[2] Exploitee.rs. https://www.exploitee.rs/.

[3] LiME: Linux memory extractor. https://github.com/504ensicsLabs/LiME.

[4] Linux kernel CVEs: Linux vulnerability tracker. https://www.linuxkernelcves.com/.

[5] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A secure and reliable bootstrap architecture. In *Proceedings. 1997 IEEE Symposium on Security and Privacy*, 1997.

[6] Arm Ltd. Cortex-M3 technical reference manual: About the NVIC. https://developer.arm.com/documentation/ddi0337/h/nested-vectored-interrupt-controller/about-the-nvic.

[7] Arm Ltd. Generic interrupt controllers. https://developer.arm.com/ip-products/system-ip/system-controllers/interrupt-controllers.

[8] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, August 2019.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the linux kernel. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 95–104, 2005.

[11] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016.

[12] Yurong Chen, Tian Lan, and Guru Venkataramani. DamGate: Dynamic adaptive multi-feature gating in program binaries. In *2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST 2017)*, 2017.

[13] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. DECAF: Automatic, adaptive de-bloating and hardening of COTS firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1713–1730, 2020.

[14] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[15] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, January 2006.

[16] Nicola Corna. me_cleaner, 2018.

[17] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 309–326, 2018.

[18] 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, 2016.

[19] Ang Cui and Salvatore J. Stolfo. Defending embedded systems with software symbiotes. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, 2011.

[20] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[21] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop (PPREW)*, 2015.

[22] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Oleinik, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *16th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, June 2021.

[23] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

[24] Phil Goldstein. Disabling USB ports: 4 ways to prevent data leaks via USB devices. https://fedtechmagazine.com/article/2017/07/4-ways-prevent-leaks-usb-devices, July 2017.

[25] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, Chaoyang District, Beijing, September 2019. USENIX Association.

[26] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 789–806, 2020.

[27] Brian Heath, Neelay Velingker, Osbert Bastani, and Mayur Naik. PolyDroid: Learning-driven specialization of mobile applications. https://arxiv.org/abs/1902.09589, 2019.

[28] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[29] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *2011 IEEE Symposium on Security and Privacy*, pages 347–362. IEEE, 2011.

[30] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[31] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*, 2019.

[32] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[33] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An application-oriented linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 20(6):1093–1107, 2004.

[34] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. In *NDSS*, 2021.

[35] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–132, 2020.

[36] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, 2018.

[37] Andy Nguyen. BleedingTooth: Linux Bluetooth zero-click remote code execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html, April 2021.

[38] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: Debloating the Chromium browser with feature subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[39] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In *2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST 2017)*, 2017.

[40] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[41] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36, 2020.

[42] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BLEEDINGBIT: The hidden attack surface within ble chips. Technical report, 2019.

[43] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[44] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 291–307, 2018.

[45] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys*, 54(1), January 2021.

[46] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. LIGHTBLUE: Automatic profile-aware debloating of bluetooth stacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[47] Qinge Xie, Qingyuan Gong, Xinlei He, Yang Chen, Xin Wang, Haitao Zheng, and Ben Y. Zhao. Trimming mobile applications for bandwidth-challenged networks in developing regions. https://arxiv.org/abs/1912.01328, 2019.

[48] Bin Xin, William N Sumner, and Xiangyu Zhang. Efficient program execution indexing. *ACM SIGPLAN Notices*, 43(6):238–248, 2008.

[49] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

Appendix A
Snapshot Registers

On ARM, we collect the general purpose registers (`R0-R15`), including the banked versions of R13 and R14, which have separate values for each CPU mode (IRQ, FIQ, SVC, Undefined, Abort, and User). In addition, to allow virtual address translation, we collect several important coprocessor registers: the `ttbr` and `sctlr` registers for virtual memory and page tables, the `dacr` register for memory domain access permission, the `daif` register for masked exception configuration, the `tpidrprw` register, which holds the current thread ID, the `vbar` register to get the exception table base, and the `scr` and `hcr` registers, which control the ARM secure/non-secure mode and hypervisor mode, respectively.

For MIPS, we currently only support the MIPS Release 1 architecture. We collect the general purpose registers (`$0-$31` and `pc`) as well as the `status`, `cause`, and `badvaddr` registers for the current CPU exception state, and `context` for the page table entry address.

TABLE VII
Performance

	Analysis Time	**Traces**	**Blocks**
Beagle Linux	6 min	54	324,987
Nuri Linux	1,196 min	104	9,495,362
RasPi FreeBSD	55 min	500	3,402,507
RasPi Linux	214 min	548	15,022,092
RasPi RiscOS	4 min	36	3,373
Romulus Linux	58 min	421	3,797,486
Sabre Linux	173 min	66	3,127,335
Sabre VxWorks	2 min	41	29,694
WRT54GL Linux	30 min	544	1,603,233
SteamLink Linux	21 min	31	1,060,850

Appendix B
Trace Analysis Performance

Aside from fuzzing (for which performance results are reported in Figure 4), the bulk of IRQDebloat's time is spent in trace analysis. It is important to note that this is an offline analysis that only needs to be done once per device, so we do not believe the analysis time is prohibitive. Nevertheless, we report it here for the interested reader. Our testbed is a dual-CPU 64-bit Intel® Xeon® X5690 @3.47GHz with 24 cores in total, and 192GB RAM. Each trace analysis is run on 16 cores in parallel.

Table VII shows the time taken, number of traces, and total number of basic blocks. In general, the analysis time required is proportional to the number of traces and total number of blocks. An exception to this is Linux on the Nuri platform; the traces for this system often diverge and re-converge, which increases the analysis time.

Appendix C
Fuzzer Pseudocode

```
# random(a,b): uniform random float in [a,b]
# randint(a,b): uniform random int in [a,b]
# rands(), patterns(), bitwins(), ints():
#   Return a list of I/O values according to
#   the patterns described in §III-D1

# Globals for use in mmio_cb
seq, iov = None, None

# Called by emulator on MMIO reads
def mmio_cb():
  global seq, iov
  if seq: # Still have some I/O values
    iov = seq.pop()
    return iov
  else:
    if consistent_io_prob > 0 and \
       random(0,1) < consistent_io_prob:
      # Use most recent
      return iov
    else:
      iov = randint(0,2**32)
      return iov

# Check for new coverage for an I/O seq
def run_trace():
  trace = []
  revert_to_snapshot()
  trigger_interrupt()
  # Execute and feed I/O values
  for n, block in enumerate(emulate()):
    trace.append(block)
    if n > MAX_BLOCKS: break
  return trace

# Generate fuzzed inputs and test if they
# produce new coverage
def fuzz(MAX_GEN):
  coverage = set()
  global seq
  seeds = [[]]
  for i in range(MAX_GEN):
    for s in seeds:
      for val in rands() + patterns() \
                + bitwins() + ints():
        seq = s + [v]
        # Get trace for this seq
        trace = run_trace()
        # Check coverage
        if new_cov(trace, coverage):
          coverage.update(set(trace))
          report(trace)
          seeds.append(seq)
```